Crawl Smart: A Domain-Specific Crawler

Prakash Hegade, Ruturaj Chitragar, Raghavendra Kulkarni, Praveen Naik, and A. S. Sanath

Abstract With billions of people using the internet, which consists of an estimate of one billion websites, is explored by an individual with a diverse need and intent. The search engines that present results to internet user queries evaluate the websites on numerous parameters to sort the links from most to least relevant. It has become a pick-and-shovel task to extract the most relevant information for a given concept or user query. Classical crawlers that use traditional crawling techniques pull irrelevant data with the relevant ones, resulting in ineffective CPU time usage, memory, and resources. This paper proposes a knowledge-aware crawling system, Crawl Smart, which learns from its own crawling experiences and improves the crawling process in future crawls. The project's key focus is a methodology that deploys a unique data structure to overcome the challenges of maintaining visited pages and finding a relation between the crawled pages after having them in the knowledge base, which helps the crawler preserve focus. The data structure design, annotations, similarity measures, and knowledge base supporting the Smart Crawl are detailed in the paper. The paper presents the results that show the comparison between the knowledge-aware crawler and the traditional crawler, assuring better results when used on large-scale data.

Keywords Annotations · Crawler · Domain-specific · Knowledge-aware · Similarity

1 Introduction

Internet has grown to an extent where any information the user is looking for, is readily available and the challenge is to extract only the most relevant information about a given concept. Be it a search engine or a user, the relevant data needs to be parsed and extracted from the humongous web. Web crawling, used mostly by search engines, are softwares that are used to discover and index websites [1]. Along with

P. Hegade (✉) · R. Chitragar · R. Kulkarni · P. Naik · A. S. Sanath
KLE Technological University, Hubballi, India
e-mail: prakash.hegade@kletech.ac.in

M. D. Borah et al. (eds.), *Big Data, Machine Learning, and Applications*, Lecture Notes in Electrical Engineering 1053, https://doi.org/10.1007/978-981-99-3481-2_25

establishing relationships between the web pages, crawlers are used to collect and process information that can be used to classify web documents and provide insights into the collected data [2].

The process of crawling and parsing also leads to the extraction of irrelevant information from the web. For example, classical crawlers adopt breadth-first mechanism [3], i.e., searching all the links of a parent link, extract both relevant and irrelevant data from the web, and hence result in wastage of CPU time, memory, and resources. To resolve these challenges topic specific crawlers, also known as focused crawlers [4] are introduced. These are better than classical crawlers in producing accurate data for the given concept.

If a user wants to crawl a website, he first needs an entry point. Back in the early days, users had to submit the website to search engines to tell them it was online. But now users can quickly build links to their website and make it visible to the search engines on the web. A crawler works with a list of initialized links called Seed URLs. These are passed on to a fetcher that extracts the content of the web page of the URLs, which is further moved on to a link extractor that parses the HTML and extracts all the links present in it. These links are passed to a store processor, which stores them, and a page filter that sends all the interesting links to a URL-seen module. The URL-seen module checks whether the URL has already been seen before in this crawling process, if not, it is sent back to the fetcher. This iterative process continues until the fetcher gets no links to fetch the content [5].

The retrieval of information is design-dependent and does not always result in styles that adapt the workflow. On the other hand, smart crawler or knowledge-aware crawler, is context-dependent. Majority of re-crawling techniques assume the availability of unlimited resources and zero operating cost. But in reality, the resources and budget are limited and it is impossible to crawl every source at every point of time. This brings up the idea of providing a mechanism to the crawler with which it can store its experience/knowledge about web pages and use it, when needed. We aim to develop and deploy the knowledge-aware crawler focusing on the context needed by the user. This is done by the integration of various modules from maintaining crawled data in an appropriate structure to finding similarities and building a knowledge base. Then, we contrast it with the traditional crawlers to know the gains in the future crawls.

This paper is further divided into the following sections. Section 2 presents the literature survey. Section 3 presents our Smart Crawl model design and deliberations; Sect. 4 presents the results and discussion; and Sect. 5 presents the conclusion.

2 Literature Survey

Internet documents contain the latest and relevant contents, which is essential to construct an encyclopedia, i.e., textual data and multimedia data such as images, videos, audio, etc. With the help of the dynamic encyclopedia, we can easily crawl through the web pages from the search engine results and find what is needed without

the user interaction in filtering and combining data from individual search results. In data science, mining [6] can be the key to finding relevant keywords from millions of web pages without having to read everything.

Irrespective of the industry, annotations tools are the key to automatically index data, synthesize text, or create a tag cloud using the most representative keywords [7]. With automatic annotation extraction, we can analyze as much data as we need. We can manually read the whole text and define key terms, but this takes a long time [8]. Automating this task allows us to focus on other parts of the project. Extraction of annotations can automate workflows, such as adding tags to web content, saving us a lot of time. It also provides actionable, data-driven insights to help make more informed decisions while crawling [9].

Manual procedures to extract statistics from textual information may be challenging for giant duties in which assets are limited, as they usually are. Computer-assisted techniques appear to be a promising alternative: Hence, researchers can complete specific tasks with a great speed. Every manual, automatic, or semi-automatic technique for analyzing textual information has its set of blessings and expenses that fluctuate depending on the venture at hand [10].

Over the period of time, crawlers on numerous criteria like, Parallel and Topical web crawlers have been designed [11–13]. General evaluation framework for topical crawlers has been discussed [14]. Measures to improve the performance of focused web crawlers have been deliberated [15]. Crawlers used in developing search engines have been surveyed [16]. Studies have been made on different types of web crawlers [17]. Behaviors of the web crawlers have been modeled [18]. Advanced web crawlers have been discussed [19]. Crawlers have been designed based on inferences and as well by contextual inferences [20, 21].

Finding helpful information on an extensively distributed internet network requires an effective search strategy. Web resources' spread and dynamic nature pose a significant challenge for search engines to keep your web content index up to date because they must search the internet periodically [22]. There are many technical challenges faced in designing a crawler [23]. Some of them are avoiding visits to the same link frequently, avoiding redundant downloading of web pages and thus utilizing network bandwidth efficiently, avoiding bot traps, maintaining the freshness of the database, bypass login pages and captchas, crawling non-textual content of HTML pages, etc. Following a broad literature survey, we would be addressing the following challenges through this project work, avoiding multiple visits to the same link, avoiding redundant downloading of web pages, and knowing whether to crawl a page or not, before opening it.

3 Crawl Smart Model

This section presents the design and deliberations of the smart crawl. Our objective is to design and develop a knowledge-aware crawler that learns the web with time, by storing the context of the web pages it visits in the form of knowledge and then using it to improve crawling in future.

3.1 Design Principles

The design principles of the crawler are:

- To design a data structure for contemporary crawler challenges,
- To annotate crawled data,
- To establish relationships among the crawled data, and
- To build a knowledge system with parsed data.

3.2 Model Design

The system model is presented in Fig. 1.

The crawler first looks for the query tag given as input, in the knowledge base. If the tag is mapped to a link, that corresponding link is used to start crawling. Else,

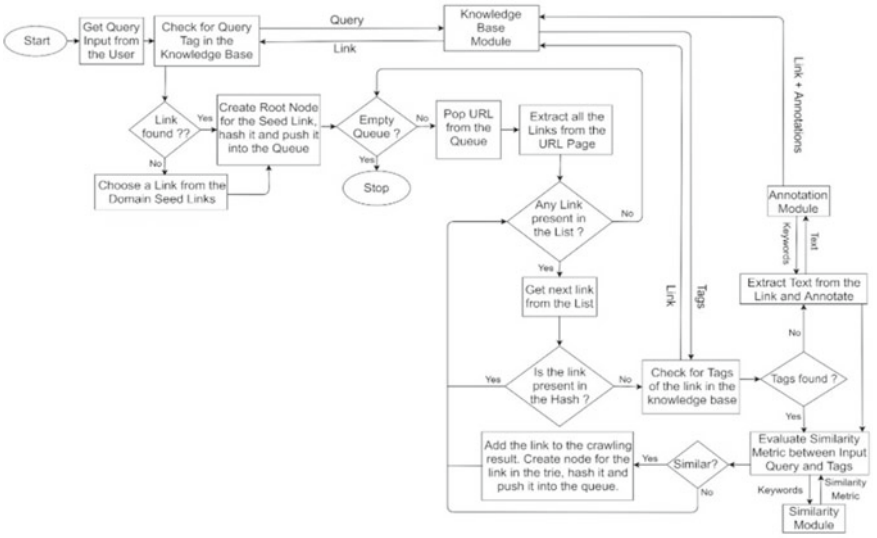

Fig. 1 Crawl smart system model

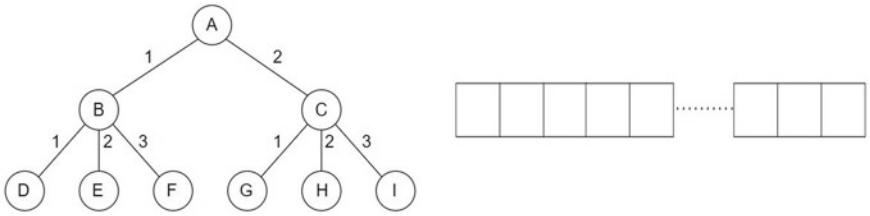

Fig. 2 Data structures to store crawled links: Trie and Hash Table

a random link belonging to the domain of the input query tag is chosen from the pool of seed links, and crawling is initialized. For every crawled link, the crawler first checks if there are any tags associated with the link in the knowledge base. This step avoids annotating links that were already visited in the previous crawling. If tags are available, it extracts the tags and uses them for the similarity metric. Else, it extracts text from the link and annotates it. Here, a copy of the tags is inserted into the knowledge base for future reference. Upon evaluating the similarity metric, if it is above the threshold, then the link is considered for the next iteration; otherwise, the link is discarded.

3.3 Data Design

The data structure designed for this work has two major components and is represented in Fig. 2.

Trie. It is a rooted tree, where a Parent Node is linked to one or more nodes, called the Child Nodes. The root node has no parent node, and the Leaf nodes have no child nodes. Apart from these properties of a tree, every child node of a Trie has a unique integer ID concerning its parent.

Hash Table. The Hash Table is a map, mapping the node value entering the Trie to a unique key generated by the hash function. The size of the Hash Table is kept dynamic.

3.4 Abstract Data Type Representation

The Abstract Data Type (ADT) Representation of the data structure can be described as shown in Fig. 3.

```
abstract typedef <<eltype>> NODE (eltype);
    abstract addChild(value)
        NODE(eltype) value;
        postcondition: childNodes == childNodes + 1
                       children[childNodes] == value

abstract typedef <<eltype>> PREFIX_TREE (eltype);
    abstract insertNode(parent,child)
        PREFIX_TREE(eltype) parent;
        PREFIX_TREE(eltype) child;
        precondition:  hash[parent] != ''
        postcondition: parent.childNodes == parent.childNodes + 1
                       parent[parent.childNodes] == child
                       hash[child] == hash[parent] + delimiter + parent.childNodes
    abstract NODE getNode(value)
        PREFIX_TREE(eltype) value;
        precondition:      hash[value] != ''
    abstract NODE getParent(child)
        PREFIX_TREE(eltype) value;
        precondition:      hash[child] != ''
```

Fig. 3 Abstract data type representation

The objective of this work is to design a data structure that can efficiently store the relationship between the crawled data. Now, the web can be seen as a hierarchy of tens of millions of web pages, resembling the Trie data structure, making it suitable to store crawled data along with their relationships. Also by the concept of hashing, we can make sure that no link is inserted more than once in the trie. The two components of the data structure interact with each other to uniquely identify a particular value placed in the Trie using a Hash key. The hash function which generates a unique hash key for every input is recursively defined as

Algorithm Hash (node)
//Input: node
//Output: hash value of the node
//Description: computes a hash key for every input 'node'
if node = root **then**
return string(0)

X ← Number of children of parent Node
return Hash(parent) + delimiter + string(X+1)

```
Algorithm getNode(key,delimiter)
//Input:        key, delimiter
//Output:       node
//Description: searches for a node with given 'key'
Initialize node to root
IDs←split the key with respect to the delimiter into a list
IDs←drop the first element of the IDs list
for id in key:
      node←idth child of node
      if node = null:
            exit for loop
      endIf
endFor
return node
```

Theorem 1 *Hash(node) hash function generates unique hash value for every key.*

Proof Every node in the Trie must be either a root node or a child node, and not both at a time. Now consider two nodes x and y from a Trie with at least two nodes.

Case 1 **x and y are directly connected**

If x is a parent of y, then
hash(y) = hash(x) + delimiter + someID ⇒ hash(y) ≠ hash(x)
If y is a parent of x, then
hash(x) = hash(y) + delimiter + someID ⇒ hash(y) ≠ hash(x)
Thus, hash(x) ≠ hash(y)

Case 2 **x and y are siblings, i.e., they have a common parent**

Let z be the parent of x and y. Now,
hash(x) = hash(z) + delimiter + ID1
hash(y) = hash(z) + delimiter + ID2
Since ID1 ≠ ID2, hash(x) ≠ hash(y)

Case 3 **x and y are not related to each other**

Let x1 be the parent of x and y1 be the parent of y.

- If x1 and y1 are directly connected, then hash(x1) ≠ hash(y1), by case 1.
  hash(x1) + delimiter + ID1 ≠ hash(y1) + delimiter + ID2
  hash(x) ≠ hash(y)
- If x1 and y1 are siblings, i.e., if they have a common parent
  hash(x1) ≠ hash(y1), by case 2.
  hash(x1) + delimiter + ID1 ≠ hash(y1) + delimiter + ID2
  hash(x) ≠ hash(y)
- If x1 and y1 are not related to each other
  hash(x1) ≠ hash(y1) {termination condition falling under case (01) or (02)}

hash(x1) + delimiter + ID1 ≠ hash(y1) + delimiter + ID2
Thus, hash(x) ≠ hash(y) for all x and y.

Thus, Hash(node) hash function generates a unique hash value for every key.

3.5 Annotations

Data annotation is similar to tagging which allows users to organize information by combining them with tags or keywords. We annotate the textual content of a web page to come up with metadata that explains the context of the web page in brief. When the same web page is encountered again, these annotations help us know the context of the page, without parsing the entire page once again. Since more than 80% of the textual data from the web is unstructured or not categorized in a predetermined way, it is extremely difficult to analyze and process them. Hence, the crawlers can extract the keywords from the web to analyze the context of the web page more effectively. Our annotation module can be seen in Fig. 4. The model uses Term Frequency–Inverse Document Frequency (TF-IDF) model that comes from the language modeling theory: TF-IDF is presented in Eq. 1.

$$W_{ij} = t_{ij} * \log\left(N / \mathrm{df}_{ij}\right) \tag{1}$$

Here, t_{ij} = term frequency score, df_{ij} = document frequency score, and W_{ij} = Weight of a tag in the text. Although a tag can have multiple words and there are resources in the modeling theory to analyze them, they are heavily time- and resource-consuming, and hence out of the scope of this work.

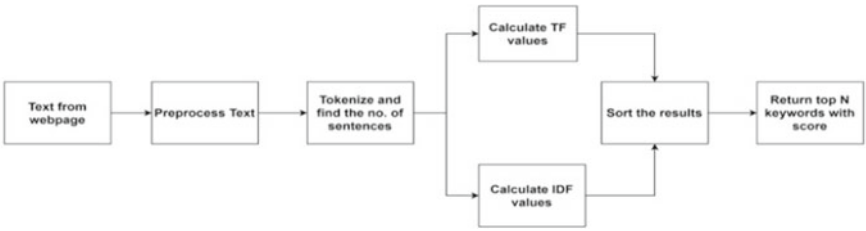

Fig. 4 Annotation module

```
Algorithm getAnnotations(text,N)
    // Input :          text, N
    // Output :         dictionary
    // Description :    dictionary of top 'N' tags from 'text' mapped to their score
    refinedWords ← getTotalWords(text)
    sentences ← getTotalSentence(text)
    tfScore ← calculateTF(refinedWords,length(refinedWords))
    idfScore ← calculateIDF(refinedWords, sentences)
    for word in tf Score:
        if idfScore[word] = 0:
            tfIdfScore[word] ← 0
        else
            tfIdfScore[word] ← tfScore[word] x idfScore[word] x 10
        endIf
    endFor
    return tfIdfScore
```

3.6 Similarity Module

The similarity module is presented in Fig. 5. The major components of this module are WordNet, spaCy, and Knowledge Base. This module has components derived from Natural Language Processing. The input to the system is from the user in the form of a tag. This tag is then matched with the annotations present in the knowledge base, if the threshold of the match is crossed then the respective link in knowledge base is returned as result, if no annotations in knowledge base match the entered tag, new crawl is initiated with the input query as a tag.

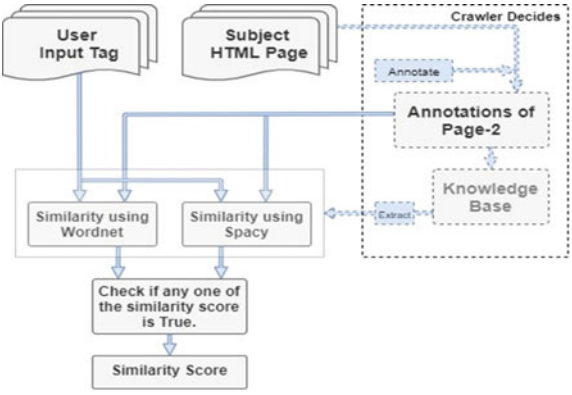

Fig. 5 Similarity module

```
Algorithm is SimilarWordnet(word, sent)
// Input :          word, sent
// Output :         Boolean value
// Description :    checks if the input tags are similar using WordNet
for sysnet in wordnet.synsets(word):
        for lemma in sysnet.lemmaNames():
                if lemma in sent :
                        return true
return false
```

```
Algorithm  is SimilarWordnet(word, sent)
    // Input :          word, sent
    // Output :         Boolean value
    // Description :    checks if the input tags are similar using Spacy
word = word1+" "+word2
tokens = nlp(word)
token1, token2 = tokens[0], tokens[1]
if token1.similarity(token2) = threshold:
        return true
return false
```

3.7 Knowledge Base

The crawler has a knowledge base where all the previously crawled links along with their annotations (tags) are maintained. Since the user provides the input query as a word, the crawler should look for links with tags matching the input query in the knowledge base. So, the tags should be mapped to their respective links. During the crawling, the crawler looks for the tags of a particular link in the knowledge base. So the links should be mapped to their respective tags. Also, a link can have more than one tag, and it needs to be mapped to each of them. Similarly, a tag can be a part of one or more links, and it should be mapped to each of them. Thus, knowledge base is a bidirectional multi-mapping. The internal storage is presented in Fig. 6.

4 Results and Discussion

The input to the system is a keyword given as a query by the user. The system needs to start with a link, sometimes also known as a seed link, which is in context to the input query, and crawl all the further links using the algorithm described in the

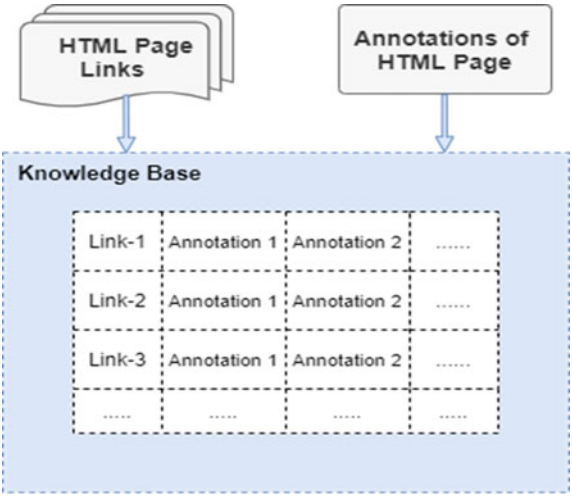

Fig. 6 The knowledge base

previous chapters. Adhering to the system capacity, power constraints, and scope, we scale down the input tests and give some relaxations in the performance metrics, according to the following assumption: user makes no spelling errors in the input query. The results of traditional and our crawler are tabulated in Table 1. We have used the website geeksforgeeks to tabulate the results as the site allows the bots to visit respective pages (rules as stated as of 01 December 2021). The system will stop crawling links once it has crawled at most 10 links.

Five more queries were considered in this test, only on the smart crawler. But in this test, two successive iterations were conducted on the same set of queries to compare the performance of the crawler in both, the absence and presence of links in the knowledge base. The results obtained were plotted as shown in Figs. 7 and 8. The time comparison can be seen in Fig. 9.

Observations: The smart crawler was able to fetch more relevant links than the traditional crawler for the same set of queries. Also, the smart crawler took an average crawling time of (67.96 ± 60.092) seconds, for the execution of a query in the first iteration, while it took an average of (2.3 ± 1.4) seconds in the second iteration. The difference observed in the average time taken to crawl the links for a particular query indicates that the presence of the context of a web page being crawled in the knowledge base has successfully reduced the crawling time of the crawler by avoiding it from going through the content of the web page again.

Table 1 Results for the input query "sorting"

Traditional crawler	Smart crawler
www.geeksforgeeks.org/sorting-algorithms/?ref=ghm	www.geeksforgeeks.org/sorting-algorithms/?ref=ghm
#main	www.geeksforgeeks.org/sorting-algorithms/
www.geeksforgeeks.org/	www.geeksforgeeks.org/recursive-bubble-sort/
www.geeksforgeeks.org/topic-tags/	www.geeksforgeeks.org/insertion-sort/
www.geeksforgeeks.org/company-tags	www.geeksforgeeks.org/recursive-insertion-sort/
www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/?ref=ghm	www.geeksforgeeks.org/radix-sort/
www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/?ref=ghm	www.geeksforgeeks.org/timsort/
www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-analysis/?ref=ghm	www.geeksforgeeks.org/pigeonhole-sort/
www.geeksforgeeks.org/analysis-of-algorithems-little-o-and-little-omega-notations/?ref=ghm	www.geeksforgeeks.org/cycle-sort/
www.geeksforgeeks.org/lower-and-upper-bound-theory/?ref=ghm	www.geeksforgeeks.org/cocktail-sort/
www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/?ref=ghm	www.geeksforgeeks.org/stooge-sort/

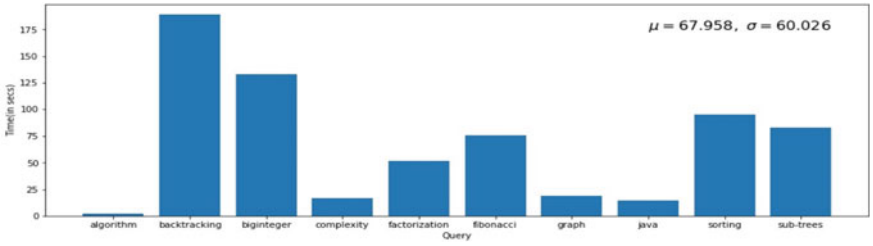

Fig. 7 Iteration 01: crawler in the absence of knowledge base

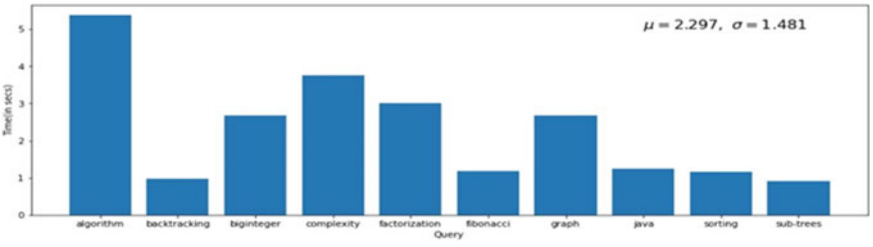

Fig. 8 Iteration 02: crawler in the presence of knowledge base

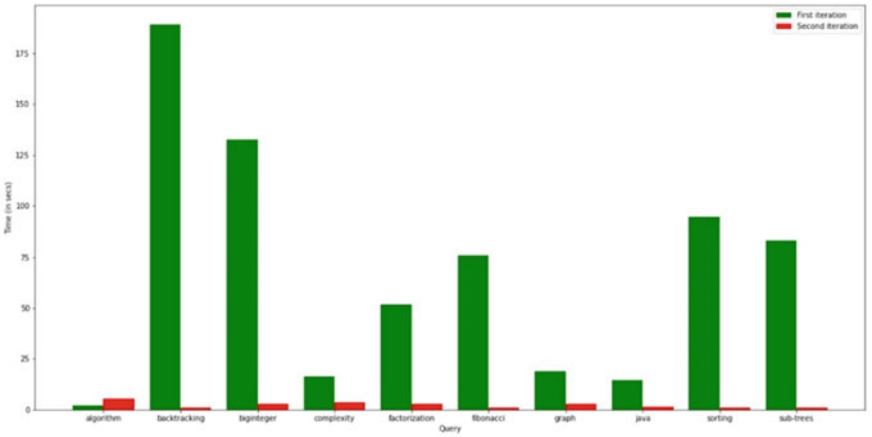

Fig. 9 Comparison of time required

5 Conclusion and Future Scope

The objective of the work is to design a crawler, which is aware of a link or a URL page and can decide whether to crawl the link or not without opening or requesting the HTML page of the link. This has been achieved through the knowledge base which acts as a memory for the crawler. Regardless of whether a new link opened and annotated is used in the current crawling process or not, it will be added to the knowledge base. This saves time in requesting, downloading, and annotating a web page that has already been downloaded in the past. Thus, the crawler learns the web gradually with time and improves its performance as more and more links get added to the knowledge base, similar to a machine learning model.

References

1. Kausar MA, Dhaka VS, Singh SK (2013) Web crawler: a review. Int J Comput Appl 63(2)
2. Najork M (2017) Web crawler architecture
3. Gupta P, Johari K (2009) Implementation of web crawler. In: 2009 second international conference on emerging trends in engineering & technology. IEEE, pp 838–843
4. Mukherjea S (2000) WTMS: a system for collecting and analyzing topic-specific web information. Comput Netw 33:457–471
5. Mirtaheri SM, Dinçktürk ME, Hooshmand S, Bochmann GV, Jourdan GV, Onut IV (2014) A brief history of web crawlers
6. Tan PN, Steinbach M, Kumar V (2016) Introduction to data mining. Pearson Education India
7. Science Direct (2019) Machine learning for email spam filtering. https://www.sciencedirect.com/science/article/pii/S2405844018353404. Last accessed 24 Sept 2021
8. Alani H, Kim S, Millard DE, Weal MJ, Hall W, Lewis PH, Shadbolt NR (2003) Automatic ontology-based knowledge extraction from web documents. IEEE Intell Syst 18(1):14–21

9. Erdmann M, Maedche A, Schnurr H-P, Staab S (2000) From manual to semi-automatic semantic annotation: about ontology-based text annotation tools. In: Proceedings of the COLING-2000 workshop on semantic annotation and intelligent content, pp 79–85
10. Cardie C, Wilkerson J (2008) Text annotation for political science research. Taylor & Francis, pp 1–6
11. Cho J, Garcia-Molina H (2002) Parallel crawlers. In: Proceedings of the 11th international conference on World Wide Web, pp 124–135
12. Menczer F, Pant G, Srinivasan P, Ruiz ME (2001) Evaluating topic-driven web crawlers. In: Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval, pp 241–249
13. Menczer F, Pant G, Srinivasan P (2004) Topical web crawlers: evaluating adaptive algorithms. ACM Trans Internet Technol (TOIT) 4(4):378419
14. Srinivasan P, Menczer F, Pant G (2005) A general evaluation framework for topical crawlers. Inf Retr 8(3):417–447
15. Batsakis S, Petrakis EG, Milios E (2009) Improving the performance of focused web crawlers. Data Knowl Eng 68(10):1001–1013
16. Deshmukh S, Vishwakarma K (2021) A survey on crawlers used in developing search engine. In: 2021 5th international conference on intelligent computing and control systems (ICICCS). IEEE, pp 1446–1452
17. Chaitra PG, Deepthi V, Vidyashree KP, Rajini S (2020) A study on different types of web crawlers. In: Intelligent communication, control and devices. Springer, Singapore, pp 781–789
18. Menshchikov AA, Komarova AV, Gatchin YA, Kalinkina ME, Tkalich VL, Pirozhnikova OI (2020) Modeling the behavior of web crawlers on a web resource. J Phys: Conf Ser 1679(3):32–43
19. Patel JM (2020) Advanced web crawlers. In: Getting structured data from the internet. Apress, Berkeley, CA, pp 371–393
20. Hegade P, Shilpa R, Aigal P, Pai S, Shejekar P (2020) Crawler by inference. In: 2020 Indo–Taiwan 2nd international conference on computing, analytics and networks (Indo-Taiwan ICAN). IEEE, pp 108–112
21. Hegade P, Lingadhal N, Jain S, Khan U, Vijeth KL (2021) Crawler by contextual inference. SN Comput Sci 2(3):1–2
22. Sharma G, Sharma S, Singla H (2016) Evolution of web crawler its challenges. Int J Comput Technol Appl 9:53–57
23. Mahajan R, Gupta SK, Bedi MR (2013) Challenges and design issues in search engine and web crawler. Ed Comm 42