Web Crawling Project Using Python and AI

By:
Joshua Souane
Elena Ajayi
Omar Shajib
Jomo Smith
Jordan Hayes
Rosa DiLiberto

Introduction:

This web crawler has been crafted with a deliberate focus on the systematic exploration and extraction of valuable information from diverse web pages, particularly St. John's University's website. Once requirements were established for the web crawler, we asked ChatGPT to create a web crawler with those requirements to generate Python code. We incorporated the code from different group members to test if the crawler worked or if additional modifications were needed. If additional modifications were needed, we would continue to prompt ChatGPT or use other resources for additional assistance. This report discusses implementing the design, analyzing crawling strategies and their effectiveness, challenges we faced when creating a web crawler, insight into Artificial Intelligence (AI) capabilities, and whether AI improved programming skills.

Web Crawler Design:

We were able to design our web crawler from the given requirements: the web crawler must; visit all web pages within the given domain, not navigate to external domains, and handle different types of web pages and links; implement at least one visiting strategy (we used preorder) and the strategy must be specified as a parameter during class instantiation; generate a corpus of text documents containing each visited page, ensure the corpus is free of HTML tags and other non essential items, and the title of each document is the title of the page visited; handle dynamic content using JavaScript engines and interpret/navigate JavaScript-rendered content; and integrate AI. This versatile web crawler, initially tested on St. John's websites, is intended for general use and theoretically compatible with most internet sites. It adeptly navigates dynamic content (e.g. website's written in React, Angular, etc) and provides a trace of crawled content, featuring text snippets and associated URLs.

The web crawler was written in Python and made liberal use of the following librares: *request, BeautifulSoup, threading, re* and *UrlLib.* The *requests* library was used to make the HTTP requests to the provided URL and provides a framework to interact with the web content.

The *BeautifulSoup* library was use to efficiently manage website scraping, serving to abstract the intricacies of data traversal away making the task simple. The *threading* library was used to run multiple tasks at once which allowed for our multithreading strategy. The *re* library was run to allow to use of regular expressions. The *UrlLib* library was used primarily used for URL parsing.

Our web crawler involvled *WebCralwer*, a basic Python class we created that instantiated with a specified URL, visitation strategy (preorder), a set of visited URLs (using Python's set to manage uniqueness and handle duplicate traversal), and a corpus stored in a dictionary which was then stored in a text filewith each web page as the title of each document. Additionally, it's important to highlight the extensive use of prompt engineering. ChatGPT chose to introduce a property for managing multi-threading, enabling the instantiation of multiple crawler instances for concurrent, thread-safe website crawling. The lock serves to prevent race conditions, ensuring exclusive access to the critical code section by a single thread at any given time.

The public methods in our code were *start_crawling()* and *get_crawled_data()*. The *start_crawling()* public method utilized a thread-safe approach with a lock, to visit a given URL, extract text content, and recursively crawl linked URLs based on a specified visiting strategy storing the results in a shared corpus while handling errors gracefully. The *get_crawled_data()* public method outputs the corpus results stored in the class property, presented as a tuple with URL and corresponding text content.

The crawler presently handles and exhibits only text content, yet it's adaptable for further customization, such as integrating sentiment analysis (the corpus retrieved can be fed into a classifier). Its current version provides a robust foundation for diverse applications. The threading capability introduced by ChatGPT enables vertical scalability, facilitating the deployment of multiple instances on a more potent machine to leverage increased processing power.

Web Crawler Implementation:

The crawler was fine-tuned for St. John's website but theoretically adaptable to others. An obstacle arose when it stalled for hours due to attempts at crawling numerous PDFs on the resource page. The solution involved instructing the crawler to disregard URLs in the resource section of St. John's site. While this adjustment shouldn't impede crawling on other sites, it may skip traversal of resource pages on sites presenting similar scenarios.

In conclusion, the developed web crawler, initially tested on St. John's website, demonstrates versatility and compatibility with diverse internet sites. The Python-based design, incorporating libraries like Request, Beautiful Soup, and Urllib, efficiently navigates dynamic content. The class design and public methods, along with prompt-engineered multi-threading, contribute to effective, thread-safe crawling with exclusive access control. The crawler, scalable and adaptable for customization, presently focuses on text content, laying a robust foundation for diverse applications. The testing and optimization phase addressed challenges encountered on St. John's site, ensuring adaptability while maintaining efficient performance. Overall, this web crawler stands as a reliable tool for web content extraction and traversal, poised for further enhancements and applications in various domains.

Visiting strategy:

In implementing the web crawler, a "preorder" visiting strategy was employed. The choice of this strategy dictates that the crawler begins exploration from the initial URL and systematically traverses the webpage's links in a depth-first manner. Upon visiting a page, the crawler extracts and processes the textual content, concurrently initiating separate threads for each discovered link. This parallelized approach enables the crawler to efficiently explore the entire website, proceeding to deeper levels before moving on to sibling links. The use of threads facilitates concurrent processing, enhancing the overall speed of data retrieval. The "preorder" strategy ensures that the crawler thoroughly explores a webpage's links before progressing to subsequent pages, thereby capturing a comprehensive snapshot of the website's content.

Challenges and Solutions:

Creating a web crawler presented several challenges. The challenges that we ran into were extracting relevant data from HTML content, runtime efficiency, and website politeness. Our first issue when creating the web crawler was extracting relevant content. Our first version of the web crawler only returned the website as the output rather than the content on the webpage. We fixed that issue by consoling Chat GPT for assistance which generated new code, but this new code led to our next extraction issue, efficient runtime.

After Chat GPT assisted in generating code that returned content, we tried to run that code. We weren't able to see the result of this code because the runtime was taking hours due to what we later found was that the crawler was running an infinite loop as well as checking PDF content. We solved this issue again by taking our code to Chat GPT. Chat GPT generated code with regular expressions that handled the PDF issue by only return content where the HTML code started with "*p*", indicating a paragraph. The code handle the infinite loop issue by generating a set of urls, therefore if a url was already in the set, it was not added to the set and ran through the crawler again. Finally, we solved the runtime issue by implementing the multithreading strategy which allowed the webcrawler to fetch multiple pages rather than one at a time to reduce overall runtime.

Another challenge can be politeness, making sure the crawler adheres to the website's policy and procedures, which can assist with ensuring the crawler is not banned or blocked from the website. We solved this issue by adding a snippet of code that respected the robots.txt part of a website as well as a crawl delay. This was an important part of our testing so that our crawler would not get blocked for days at a time.

Insights on how AI capabilities were integrated and their impact:

Leveraging Large Language Models (LLMs) like ChatGPT and Google Bard is essential for this project. These models can interpret web crawling prompts and generate outputs tailored to our needs. We opted for ChatGPT due to its superior performance compared to Bard and its longer history of existence, including being trained on the data most relevant to our project. Additionally, we began with foundational inquiries, such as asking ChatGPT about the prerequisites for a web crawler. We also solicited ChatGPT's assistance in optimizing our web crawler code, as demonstrated in the example below:

Optimizing Web Crawler

March 5, 2024

Anonymous

you are an expert at creating and refining web crawlers

I already have a web crawler, but I need help optimizing it. One issue I'm running into is the runtime on the webcrawler. To solve this issue, I limited the max depth to 10, so that it only visits 9 children websites, the moves on. Is there another solution to this issue so that my webcrawler can crawl the entire domain

ChatGPT

Optimizing a web crawler involves several strategies to improve its efficiency and runtime performance. Limiting the maximum depth is one approach, but it can prevent the crawler from fully exploring a domain. Here are some suggestions to optimize your web crawler:

1. **Implement Crawl Delay**: Respect robots.txt directives by implementing a crawl delay. This ensures that your crawler doesn't bombard a website with requests too quickly, which can strain the server and potentially get your IP banned.
2. **Use Multithreading or Asynchronous Processing**: Implementing multithreading or asynchronous processing can significantly improve the speed of your crawler by allowing it to fetch multiple pages

Here, we are faced with an issue with the code: the web crawler's runtime. Although we found a solution to it by limiting the maximum depths, we asked if there was an alternative solution to it. This is what ChatGPT had to say:

Anonymous

I think multithreading/asynchronous processing is the best solution. Before I send you my code, I want you to assure that the webcrawler will implement multithreading as well as only visit each webpage once so that there is not infinite loop

ChatGPT

Sure, multithreading or asynchronous processing is indeed a promising approach for optimizing web crawlers. It allows the crawler to fetch multiple pages concurrently, thus reducing the overall runtime. Additionally, to ensure that each webpage is visited only once and to prevent infinite loops, we can keep track of the URLs that have been crawled using a set or a database.

Please feel free to share your code, and I'll be happy to assist you in implementing multithreading and ensuring that each webpage is visited only once.

We looked at the points and utilized multithreading processing for faster code runtime. ChatGPT also asked for our code so that it can implement multithreading processing to our code:

ChatGPT

Thank you for providing your code. Let's integrate multithreading into your web crawler to improve its efficiency. We'll use Python's `threading` module to achieve this. Additionally, we'll ensure that each webpage is visited only once by maintaining a set of visited URLs.

Here's the modified code with multithreading and ensuring unique URL visits:

python Copy code

```python
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse
import threading

# Create webcrawler class
class WebCrawler:
    def __init__(self, start_url, visiting_strategy='preorder'):
        self.start_url = start_url
        self.visiting_strategy = visiting_strategy.lower()
        self.visited_urls = set()
        self.corpus = {}
        self.main_domain = urlparse(start_url).netloc
        self.lock = threading.Lock()  # Lock for thread-safe access to shared data
```

Here is the code ChatGPT provided; this way, we have managed to reduce the code's runtime so that the crawler visits multiple pages simultaneously. Even when we don't understand specific terminologies related to web crawling, we ask ChatGPT right away. We also requested ChatGPT to generate a completely new web crawler code so that we can run both, compare them for differences, and determine which one works best.

Impact of utilizing AI:

The goal of using LLM is to understand our web crawler code better. AI works best by providing a step-by-step response based on your last prompt. ChatGPT AI allowed us to understand concepts such as multithreading, natural language processing, the ‘TextProcessor’ class, and its uses on our code. By doing so, we were able to significantly reduce the runtime of our web crawler code and minimize errors simultaneously.

Did the use of AI improve your programming skills? Elaborate on this answer.

Incorporating artificial intelligence (AI) techniques into developing a web crawler has enhanced our programming skills in various ways. For instance, we learned to utilize tools like parsers, which are instrumental in parsing HTML content to extract data from websites. Through AI-guided exploration, we deepened our understanding of parsers and their application.

Furthermore, AI has exposed us to best practices and procedures in web crawler development, thereby refining our coding techniques, fostering exposure to novel concepts, and nurturing critical thinking abilities. Lastly, AI aids in creating code more efficiently and accurately, facilitating learning about programming languages, libraries, and coding patterns.

Conclusion:

In conclusion, the development and implementation of this web crawler executed the tasks of designing and implementing different types of tools, challenges, and solutions associated with web crawling, highlighting the importance of scalability, politeness, and error handling. We also discussed the impact of AI by using different types of prompts, insight into AI capabilities and whether AI improved our programing skills.

Despite our challenges in developing a web crawler, we have witnessed the potential positive outcomes of creating a web crawler. When we learn how to utilize technology, policies and procedures, we are able to capture a web crawler’s potential and possibly be able to capitalize on its use.