Project Report on Meta Career Crawler with Puppeteer

Shiyi Wang

College of Computing, Georgia Institute of Technology

CS 4675: Advanced Internet Computing Systems and Application Development

Dr. Ling Liu

February 4, 2022

Project Report on Meta Career Crawler with Puppeteer

INTRODUCTION

Three days ago, with the project deadline approaching, my initial thought was to create a simple static blog page crawler with Scrapy. However, inspired by my project team, I decided to go beyond my comfort zone and try to scrape some dynamic pages. Therefore, I decided to build a Meta Career Crawler that crawl the latest job openings on https://www.metacareers.com/.

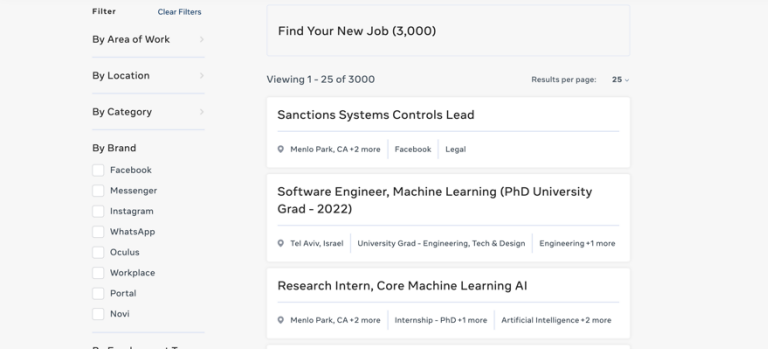


Figure 1 Meta Career Index Page

With much research into the state-of-the-art open-source crawlers, I came across Puppeteer. According to Puppeteer GitHub Page, Puppeteer is a Node library which provides a high-level API to control Chrome or Chromium over the DevTools Protocol.

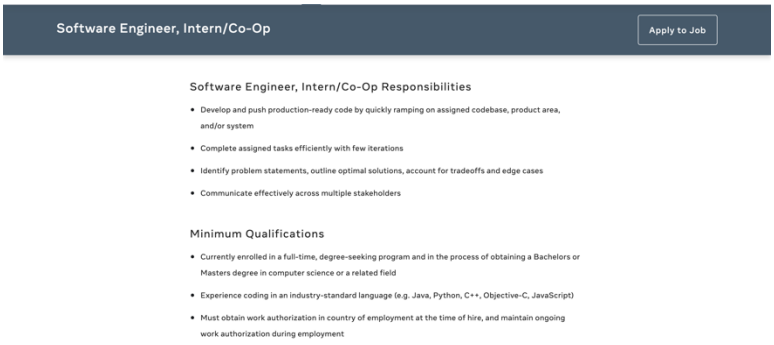


Figure 2 Meta Career Job Page

Before we dive into the design, I would like to share some thoughts on the website structure and explain the problem domain in detail. The index page is a job list consisting of 25

jobs. If we click into a job, we can see the title, locations, job descriptions, minimum requirements, and other information.

Implementation-wise, we need to click into each single subpage, extract the information we need, store as an JSON object, and go to the next one. Once the entire index page is done, we go to the next index page. Finally, I want to put all JSON objects together into the web archive. To better illustrate my idea, I plot the pipeline below.

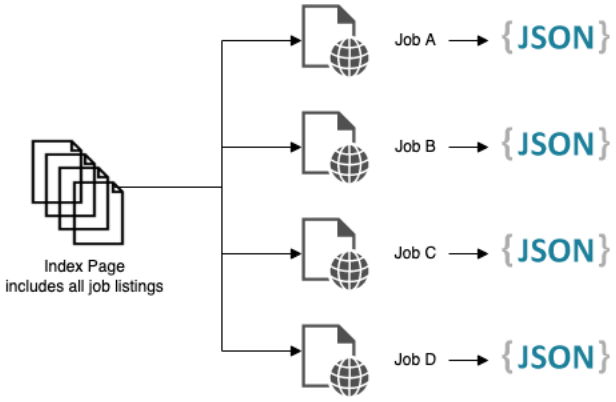


Figure 3 Implementation Idea in Abstract

CRAWLER DESIGN

Setup Module

I select https://www.metacareers.com/jobs/?page=1#search_result as the seeding URL to initialize my web crawler. Since the search page is dynamic, I use the `pg` variable to propagate through different pages. Besides, I restrict the total number of the scraped pages by setting `COUNT_LIMIT` to 1000. The reason why I set a fixed value is that Meta has an algorithm that requires the user to log in if the crawler reaches a certain limit.

```
const puppeteer = require("puppeteer");
const fs = require("fs");
const getPageUrl = (pg) => `https://www.metacareers.com/jobs/?page=${pg}#search_result`;
const COUNT_LIMIT = 1000;
```

Figure 4 Setup Module Code

Scrape Module

First, we need to extract every single URL representing a job on the search page. We can extract the CSS Selector from Chrome Developer Tools. By utilizing the puppeteer library, I put all URLs in an array named `urls`.

```
let results = [];
  const urls = await page.evaluate(() => {
    const allElements = document.querySelectorAll(
      "#search_result > div._8tk7 > a"
    );
    let urls = [];
    for (const ele of allElements) {
      urls.push(ele.href);
    }
    return urls;
  });
```

Figure 5 Index Page URL Extraction Code

Next, we can extract the keywords and subjects that we want. The first attribute I want is location. In different pages, the locations are demonstrated in different manners. Some jobs are only available in one location, but others may be available in multiple locations. For example, Software Engineer, Intern/Co-Op has location availability in 14 cities. In this case, a user needs to click on "+13 more" button to see all options. Therefore, I mimic the behavior by utilizing `page.click()` to attempt finding the button and retrieve the expanded text.

```
let locations = "";
try {
  locations = await page.$eval(
    "#careersContentContainer > div > div._9ati > div > div >
div._25w_._69fb._31bb._25w-._1icm._1ikx._1im1 > div._9atf > div > div > div._97fe._6hy- > div >
span",
    (el) => el.innerText
  );
} catch (ignored) {}

try {
  await page.click("#showLocationsButton");
  locations = await page.$eval("#locations", (el) => el.textContent);
} catch (ignored) {}
```

Figure 6 Locations Extraction Code

Similarly, I choose to extract other features including titles, responsibilities, and minimum qualifications as shown below. In this way, I can output a JSON object to store in my web archive.

```
const res = await page.evaluate(
  (locations, url) => {
    const title = document.querySelector(
      '#careersContentContainer > div > div._9ati > div > div >
div._25w_._69fb._31bb._25w-._1icm._1ikx._1im1 > div._9atj > div._9atb > div._9ata._8ww0'
    )
    const job_description = document.querySelector(
      '#careersContentContainer > div > div._3gel._3gfe._3gef._3gee._8lfv._3-8p._8lfv._3-8p >
div._25xa._69fb._31bb._3gek > div > div > div._8muv > div:nth-child(1) > div:nth-child(3) >
div._h46._8lfy._8lfy > div > ul > div:nth-child(1) > li > div._38io._30jd._9aou > div > div'
    )
    const min_req = document.querySelector(
      '#careersContentContainer > div > div._3gel._3gfe._3gef._3gee._8lfv._3-8p._8lfv._3-8p >
div._25xa._69fb._31bb._3gek > div > div > div._8muv > div:nth-child(1) > div:nth-child(4) >
div._h46._8lfy._8lfy > div > ul > div:nth-child(1) > li > div._38io._30jd._9aou > div > div'
    )

    return {
      title: title ? title.innerText : '',
      locations,
      job_description: job_description ? job_description.innerText : '',
      min_req: min_req ? min_req.innerText : '',
      url,
    }
  },
  locations,
  url
)
```

Figure 7 Feature Extraction Code

Finally, I want to find how many pages contain either “Engineer” or “Manager” as keywords. Since I have extracted these properties above, I can use a simple conditional statement to filter out the needed ones as shown below.

```
if (res.title.includes('Engineer') || res.title.includes('Manager')) {
  numOfKeywords += 1
}
```

Figure 8 Keyword Logic Code

Evaluation Metrics Module

The evaluation metrics are incorporated as console logs. I update the corresponding count, page number, time elapsed, and number of keywords in each page. Besides, I output them into a .csv file and also visible in terminal.

```
const elapsedTs = (Date.now() - startTs) / 1000
console.log(
  `[Page ${pageNum}: ${count}/${COUNT_LIMIT}: ${(
    Math.round(elapsedTs * 1000) / 1000
  ).toFixed(3)}s elapsed, ${(
    Math.round((elapsedTs / count) * 1000) / 1000
  ).toFixed(3)}s/item, ${numOfKeywords} keywords detected] Scraping ${url}`
)
history.push([elapsedTs, count, numOfKeywords, url])
```

Figure 9 Evaluation Metrics Module Code

```
[Page 85: 894/1000: 3124.257s elapsed, 3.495s/item, 514 keywords detected] Scraping https://www.metacareers.com/v2/jobs/796228031045085/
[Page 85: 895/1000: 3127.740s elapsed, 3.495s/item, 515 keywords detected] Scraping https://www.metacareers.com/v2/jobs/655838392047848/
[Page 85: 896/1000: 3131.525s elapsed, 3.495s/item, 516 keywords detected] Scraping https://www.metacareers.com/v2/jobs/901330390551814/
[Page 85: 897/1000: 3134.474s elapsed, 3.494s/item, 517 keywords detected] Scraping https://www.metacareers.com/v2/jobs/219076483743891/
[Page 85: 898/1000: 3138.038s elapsed, 3.494s/item, 517 keywords detected] Scraping https://890027078355743/
[Page 85: 899/1000: 3142.494s elapsed, 3.496s/item, 518 keywords detected] Scraping https://www.metacareers.com/v2/jobs/1533049130405169/
[Page 85: 900/1000: 3146.381s elapsed, 3.496s/item, 518 keywords detected] Scraping https://www.metacareers.com/v2/jobs/1317294452117098/
[Page 86: 901/1000: 3152.861s elapsed, 3.499s/item, 519 keywords detected] Scraping https://www.metacareers.com/v2/jobs/992449964983104/
[Page 86: 902/1000: 3156.177s elapsed, 3.499s/item, 519 keywords detected] Scraping https://317373120404837/
```

Figure 10 Console Output Sample

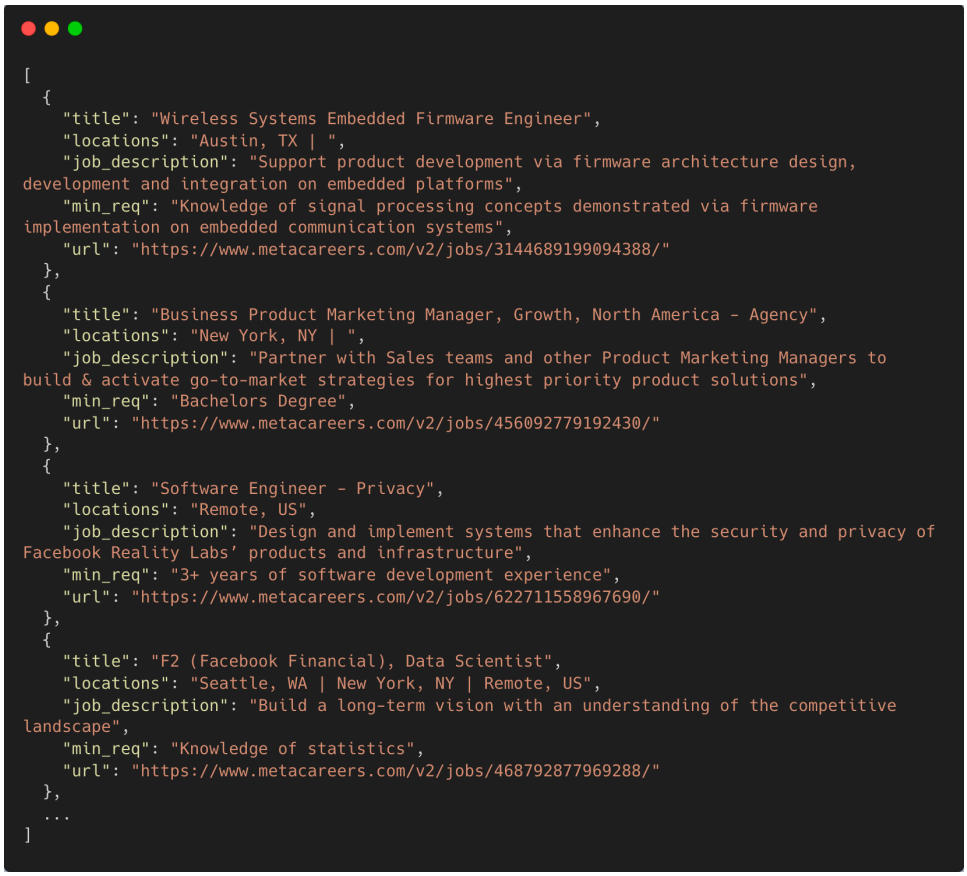
Export Module

The export module consists of two parts. The first part is history.csv which includes the statistics for evaluation, including count, page number, time elapsed, number of keywords, and the scraping URL. The second part is the web archive output. I choose to use a JSON file to store all objects for access. In the future, I can further export them into a database if needed.

```
const csvHistory =
  'Timestamp,Count,NumKeywords,URL\n' +
  history.map((row) => row.join(',')).join('\n')

fs.writeFileSync('history_final_cont.csv', csvHistory)
fs.writeFileSync('meta_final_cont.json', JSON.stringify(results, undefined, 2))
```

Figure 11 Export Module Code



```
[
  {
    "title": "Wireless Systems Embedded Firmware Engineer",
    "locations": "Austin, TX | ",
    "job_description": "Support product development via firmware architecture design,
development and integration on embedded platforms",
    "min_req": "Knowledge of signal processing concepts demonstrated via firmware
implementation on embedded communication systems",
    "url": "https://www.metacareers.com/v2/jobs/3144689199094388/"
  },
  {
    "title": "Business Product Marketing Manager, Growth, North America - Agency",
    "locations": "New York, NY | ",
    "job_description": "Partner with Sales teams and other Product Marketing Managers to
build & activate go-to-market strategies for highest priority product solutions",
    "min_req": "Bachelors Degree",
    "url": "https://www.metacareers.com/v2/jobs/456092779192430/"
  },
  {
    "title": "Software Engineer - Privacy",
    "locations": "Remote, US",
    "job_description": "Design and implement systems that enhance the security and privacy of
Facebook Reality Labs' products and infrastructure",
    "min_req": "3+ years of software development experience",
    "url": "https://www.metacareers.com/v2/jobs/622711558967690/"
  },
  {
    "title": "F2 (Facebook Financial), Data Scientist",
    "locations": "Seattle, WA | New York, NY | Remote, US",
    "job_description": "Build a long-term vision with an understanding of the competitive
landscape",
    "min_req": "Knowledge of statistics",
    "url": "https://www.metacareers.com/v2/jobs/468792877969288/"
  },
  ...
]
```


Figure 12 Web Archive Sample

Testing Module

To ensure the success of the mass crawling process, I created a standalone page testing module named `singlePageEvaluator` that allows me to input a single page URL and it logs a JSON object in the console that I am looking for. Besides, I also made a `metaCrawlLite` that crawls only 5 jobs to test the functionality.

```
(base) scott_wang@Shiyis-MBP mataCrawl % node singlePageEvaluator
{
  title: 'Manager, Production Engineering',
  locations: 'New York, NY',
  jd: "Support and lead engineers working on Facebook's products and services, at different layers of the stack, on
challenges related to scalability, reliability, performance and efficiency of systems",
  mr: '4+ years of direct management experience in a technology role'
}
```


Figure 13 SinglePageEvaluator Output

CRAWLER STATISTICS

In total, I ran two trials of experiments. The first trial goal is 1200 pages and the second is 1000 pages. Here, I plotted the crawled ratio, crawl speed, and crawl frequency graph to evaluate the performance of my crawler.

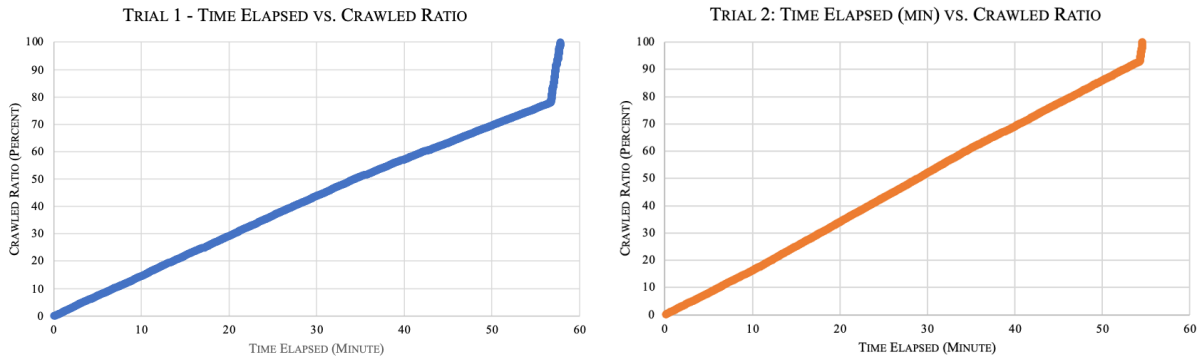


Figure 14 Time Elapsed vs Crawled Ratio Comparison

The figures above depict the relationship between time elapsed in minute versus the crawled ratio out of 1200 pages in trial 1 and 1000 pages in trial 2. The trend is general linear. This indicates that there is no outstanding difference in time between every pages. The time crawler on each page is essentially similar.

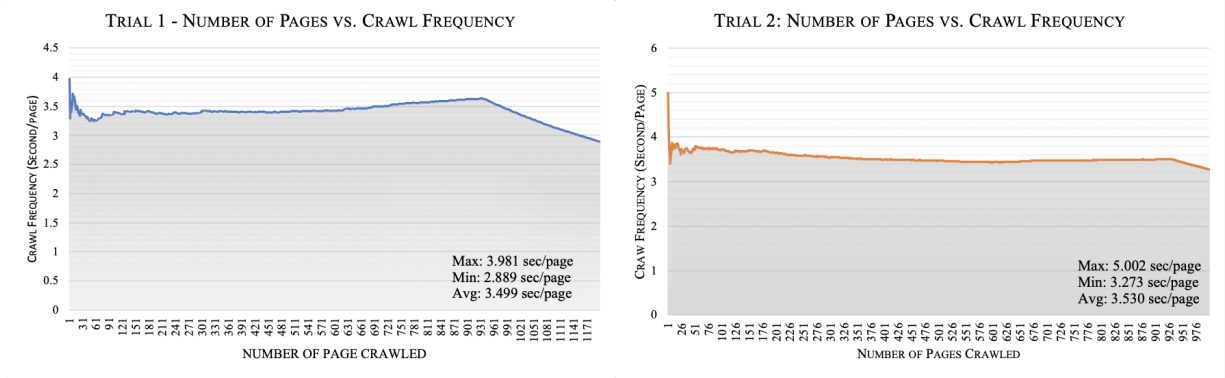


Figure 15 Number of Pages vs Crawl Frequency Comparison

The pair of figures above show the changes in crawl frequency, defining as second per page, as number of pages crawled grows. We can notice that the initial time spent on each page is high. This is because the initialization to load library, function calls, and modules takes the

majority of time. However, as process proceeds, the curve balanced out the initial overhead and stabilized in the end. In both trials, the frequency is stabilized as an average of 3.5 seconds per page (3.499 sec/page and 3.530 sec/page in either trial).

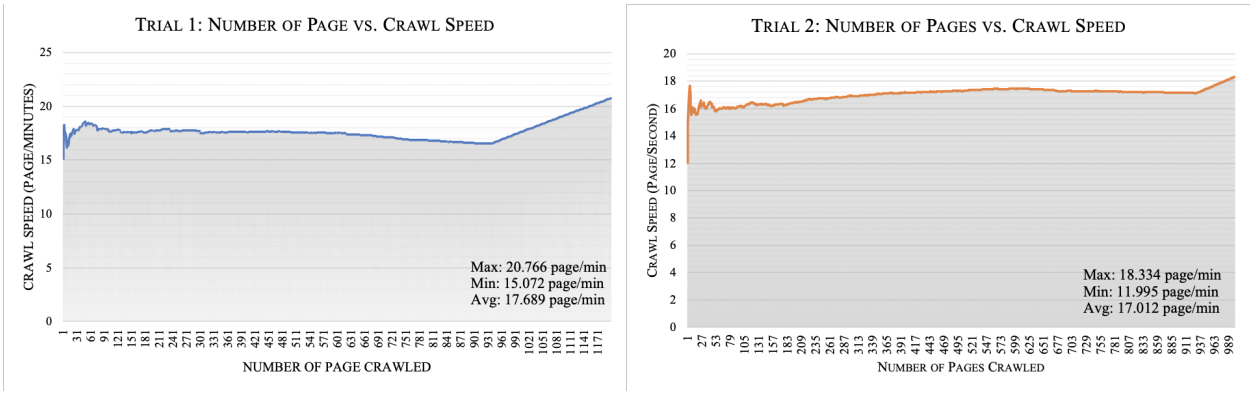


Figure 16 Number of Pages vs Crawl Speed Comparison

Finally, the graphs above show the crawl speed changes along the crawling process. The graph is intuitive and logically if we compare with the frequency graphs. One thing notable is that in the second trial, the max and min speed are both lower than the first trial. I deduce that it could relate to the performance of my local machine where the CPU usage rate is higher. Even though, the average crawl speed performance is in general consistent, around 17 pages per minute.

Discussion

Discussion on Open-Source Crawlers

There are two types of open-source crawlers. One is static and the other is dynamic.

Crawlers that rely on basic queries to HTML files are often efficient. However, when websites are constructed using contemporary frontend frameworks like AngularJS, React, and Vue.js, it occasionally captures empty bodies. For example, when I try to crawl the same webpage using Scrapy, it returns a GET error in HTML file. This is because Meta utilizes React as its framework for development where its content changes dynamically.

On the other hand, to handle scraping tasks on dynamic webpages, common crawlers include PhantomJS and Selenium. PhantomJS has deprecated and users are switching to Headless Chrome as an alternative (Puppeteer, 2017). Selenium is a cross-browser platform that operates on Chrome, Safari, Internet Explorer, and other browsers (Puppeteer, 2017). In this scenario, I do not need crawler to provide cross-browser support.

In a nutshell, Selenium is focused on cross-browser automation, and it is a single standard API that works across different browsers (Puppeteer, 2017). Puppeteer is focused on Chromium, and it demonstrates enhanced functionality and dependability (Puppeteer, 2017).

Figure 17 Comparison between Puppeteer, Selenium, and Scrapy

Lesson on JavaScript Asynchronous Events

The major technical issue I encountered was the conflict between `forEach` and `await` functions. If we are reading in sequence, we cannot use `forEach` because `await` suspends the current function evaluation, including all control structures. Instead, a for loop can fully address the issue. Another lesson I learned was related to cookies. In order to avoid login request, I turn on the incognito mode on Puppeteer which smoothened the process.

Crawl Speed Prediction

In the statistics section, we notice that the crawl speed stabilizes at 17 pages per minute in both trials. If we need to crawl 10 million pages, we need in total of

$$10{,}000{,}000/17 \; = 588235 \; minutes = 1.12 \; years$$

Similarly, if we need to crawl 1 billion pages, we will need

$$1{,}000{,}000{,}000/17 \ = 58823529\ minutes = 111.9\ years$$

The number is shocking. However, considering we are scraping dynamic pages and the response time of Meta Career website is inherently slow, this is reasonable. To optimize the solution, we can utilize the page functionality of Puppeteer by opening multiple pages (meaning multiple crawlers) running in parallel. In this way, we can cut the running time exponentially.

Keywords and Further Optimization

Finally, in the first trial, we detected 534 keywords out of 1200 links and 535 out of 1000 links in the second trial. In total, I can build a keyword database of more than 1000 URLs. One more potential optimization I propose is that Puppeteer provides low to mid-level APIs for my crawler. Puppeteer nevertheless was originally intended for automation testing. If we can build a higher-level crawler library based on Puppeteer, we can have more controls on what features to implement in order to satisfy our crawling needs.

References

Carbon. (n.d.). Carbon.now.sh. Retrieved February 4, 2022, from https://carbon.now.sh/

LearnWebCode. (2021, July 26). *Web Scraping with Puppeteer & Node.js: Chrome Automation*. Www.youtube.com. https://www.youtube.com/watch?v=lgyszZhAZOI

Meta. (2022). *Facebook Careers | Do the Most Meaningful Work of Your Career*. Facebook Careers. https://www.metacareers.com/

Puppeteer. (2017, May 7). *Puppeteer*. GitHub. https://github.com/puppeteer/puppeteer