So you want to build a search engine?

Benji Altman

May 10, 2018

Contents

Abstract

There are many different types of search engines for many different use cases. In this paper will investigate how one may make a search engine for the Internet or a similar entity. In doing this we need some way of discovering possible results, matching these results to a query and ranking possible results. This paper deals largely with the discovery and more specifically a process called web crawling.

1 Who this paper is for

This paper assumes the reader has no special knowledge of Mathematics, Graph theory, or the workings of the Internet. The paper does however use very common terminology from Mathematics and Graph theory without providing definitions. If a reader is unfamiliar with certain terms they are encouraged to look them up as otherwise the paper will spend to much time describing mathematical concepts that are outside the scope of this paper.

2 Introduction

So you want to build a search engine? Now there are a couple different types of search engines and not all search engines need to work the same, however there are two steps that are absolutely necessary for any searching to make sense. First you are going to need to have some set A to search through and second you will need a search function $s : Q \to \mathcal{P}(A)$[1] where Q is the set of all possible search queries. If we step away from mathematical formalities, this makes sense, as we simply are finding a way to match a search query to set of possible results.

Now what one might notice is that none of this relates to ranking the results, and this is obviously an important part of any modern search engine. The problem of ordering may, in fact, be the most interesting part of a search engine. This paper however will spend very little time talking about ordering a search result as Page Rank is already well understood and after that most of the complexity really has more to do with understanding the query which we will cover when talking about our search function.

Now this paper will mostly focus on the discovery of set A and assumes and Internet like structure to A. That is to say that A is a large set with a directed graph structure to it. Additionally A is not static, edges in A's graph structure may change, members of A may be created or destroyed, as may the content of those members. Finally A is not initially known to us.

Now while I try to keep this paper in the abstract and I talk about A as if it's a set and does not have to be the Internet, I think it's important to talk about some of the tools that we have built to help us work with constrains the Internet put on us. This is because the Internet is so well known and the thing that most people have experience searching. Leaving out practical information, while it may take away from the abstractness, it is the authors opinion that it is too important to leave out. In this vein we will be focusing on Google's architecture as it is so well known and information is readily available on it.[2]

3 The Internet

3.1 How it works

The Internet itself is a somewhat complex entity and a lot goes into making it work. For this we don't need to know everything but it would be good to have a rough idea of what happens when you go to a web page.

Lets say you open your web browser and go to the `www.google.com`. Now what happens is your computer sends a message out to `www.google.com` and a response is sent back to you. The web browser that you use (ie: Mozilla Firefox, Internet Explorer, Google Chrome, ...) will then display that message in a way that makes most sense.

The message that is sent to `https://www.google.com` ins't terribly complicated, however we aren't going to deal with its entire protocol, instead we will pretend that the message is simply the URL that you type in. Now you might see a somewhat more complicated URL then just `https://www.google.com`, maybe you'll see `https://www.google.com\mail` or even more complicated looking like

```
https://www.google.com/search?ei=k1TuWuoH0pCaBcvOuIgC\&q=hello+world\&oq=hello+
    world\&gs\_l=psy-ab.3..0j0i20i264k1l2j0l5j0i10k1j
    0.2636.3822.0.3932.11.8.0.0.0.0.252.676.2-3.3.0....0...1c.1.64.psy-ab
    ..8.3.676...46j0i67k1j0i46k1j0i131k1.0.2UY03D\_JQgE
```

we can think of this as simply sending that message to `www.google.com`, and them sending back a reply. Now you can see the actual text of the response that is sent back to you, however what is important to note is that this is simply a response. This means that `https://www.google.com` can send me any response it likes and it doesn't need to look like a web page and it does not need to be consistent. This means that there really ins't information on the Internet, just URLs and servers that give responses.

When we make a Internet search engine we are making some implicit assumptions about how websites like `https://www.google.com` are going to act. First we assume that most urls will respond with relativly consistent responses. What is meant by this is that if I have a URL x that produces response r_0, the first

[1] $\mathcal{P}(A)$ is the power set of A.

[2] This is not fully accurate, there is some very good information on Google's inner workings that comes from before 2000, however much of of modern Google's inner workings are kept as trade secrets or simply not released.

time I made a request for it, then when I send another request to x and get response r_1, then r_1 should be similar in content to r_0.

Now in practice, as of writing this paper, most responses (at least for important web pages) tend to be fairly static, often being exactly the same for a length of time and not changing by a large amount when they do change. For example think of the website `https://www.wikipedia.com` that acts as a communally sourced encyclopedia. Now as anyone may make a change to any particular page on this site you might think it to be rather volitile, however these changes tend to be very minimal and wont completley change the content of the page. When we talk about crawling we will useing and working within this constraint that pages change, but only gradually.

For this section we have thought of web pages simply a responses to a URL. Now that we have justified that these responses are mostly consistent, we will think of them rather as pages.

3.2 Some terminology

In order for us to understand how a search engine like Google operates, there is some terminology pertaining the Internet that we must understand. Admittedly the definitions here are not rigorous, however the quick overview of some ideas is all that is needed for our discussion.

- **Website** - A website might be best though of as a server. Each website has a single computer[3] that deals with incoming requests and makes responses. A website is denoted by `www.google.com`, anything that is appended after that is still on the same website, that is that `www.google.com` is still sending the responses.
- **Web page** - A web page is a single page or URL on a web site. More precisely a web page is the response given to you by a website given a specific URL. In this we will think of the URL as being the name of the web page while the page itself being subject to change. That is to say if a page was to be moved from URL a to URL b we would simply see it as a's web page changing to not exist and b's web page changing to whatever a used to be.
- **URL** - The URL is the 'name' of a webpage and tells us how to get there. We have already seen some example URLs like `https://www.google.com`. We see URLs come in a a few flavors, we will break them into two catagories.
  - **absolute URL** - This a URL that has the whole name in it like the ones we have seen. This contains all the information we need to get to the specific page. These often look something like `www.domainName.topLevelDomain/optionalExtraStuff`.
  - **relative URL** - This a URL that works within a website. If I'm at `www.example.com/index` and I go to the URL `/home` there is an agreement that this link really refers to `www.example.com/home`.

  If which type of URL is not specified it should be assumed that we are referring to an absolute URL.
- **404 Error** - Sometimes web pages are removed. When we try to go to a URL that is not set to respond with anything, the website will respond with a 404 error. This becomes important as we may find that some pages that used to exist no longer do, and these must be handled in a meaningful way.
- **Link** - A link is a URL (either absolute or relative) contained within a web page. A link's URL may be invalid, this we call a **dead link**. We will active links (ones that aren't dead) as the out edges in our graph model of the Internet
- **Content** - The content of the page is the actual text and information that someone viewing it might see. This will be very important for matching a search query with a result.

[3] In practice the single computer is actually multiple computers

4 Discovery

You want to create a search engine that searches through some set A, however you do not have all of A, so you need some way to discover as much of A as you can. Now for this we are going to assume you already know some $S \subset A$ and that A has a directed graph structure to it that you may follow to discover new nodes. In order to do this we will need a graph searching algorithm. There are two basic algorithms available: Depth-First search and Breadth-First search.

A depth-first search starts with a single vertex v_0, it finds all children of v_0 and then repeats the process recursively on them one at a time. Notice that if v_1 is v_0's 'first' child then we will find each v_1's children, and then repeat on v_1's first child continuing in this fashion until there are no more children and we are able to work back up to v_0's second child. This is like searching by clicking the first link you find on a page until there is no links on some page and then going back to the last page and clicking its second link. The obvious problem with this search is that it loop infinitely and to deal with this we make the slight adjustment that we never go to the same vertex twice.

A breadth-first search starts with a seed set S and finds the neighborhood of S. In this context we define the neighborhood of a set, S, to be all verticies $v \notin S$ for which there is some vertex $s \in S$ with an edge to v. In a breadth-first search, once we find the neighborhood of S we then repeat the process on the union of S with its neighborhood and continue recursively until the neighborhood of S is empty, at this point we have discovered as much of our graph as possible given our seed set. This is like going to a page and clicking all of its links and opening them in a new tab, then going through each tab and continually opening links in new tabs.

The choice of which search to use is very situational and has much to do with the structure of the graph you are searching. Additionally with more information about the graph you may be able to do some more complicated algorithms that are more suited to your needs, or some combination between the above algorithms.

4.1 On the Internet / Building a Web Crawler

There are quite a few useful results that we already know about the structure of the Internet that will help us to explore it. The problem with the Internet however is that it is so large that no one could ever hope to discover all of it, so instead we must find some way of trying to get the most valuable parts of the Internet discovered.

4.1.1 A First Web Crawler

The process we are talking about is often referred to as web crawling and a program that does this is called a web crawler or spider. Our first idea for building one might be as follows:

1. Start with some set of URLs S as our seed.
2. Enqueue each element of S to a queue Q.
3. While Q is not empty do the following:
   (a) Dequeue a URL from Q, let us call this URL u.
   (b) Make a request to URL u.
   (c) When a response is sent back, find all URLs (either absolute or relative) that one may find in that response, convert all relative URLs to absolute URLs. We will refer to this set of discovered URLs as D.
      - On a webpage there are certain places to look, for example an anchor tag may be a link to another page.
      - One may also use a regular expression to try and find things anywhere on the page that look like URLs. Even if our system is perfect sometimes pages go away so we should expect not all URLs to be valid.

In practice we may want to use some or all of the information from the web page. This will be important in later sections, but for now just realize that we can keep track of the responses we get and deal with them how we choose.

(d) For each URL $d \in D$, if $d \notin S$ then add d to S and enqueue it onto Q.

Now this web crawler works perfectly well, however it's going to be incredibly slow. The first step in optimizing this is always looking at what the slowest part is. Without even testing this program there is going to be a clear winner. When we measure the amount of time single instructions take in a computer we usually operate on the order of nanoseconds ($10^{-9}s$), whereas when we measure about the amount of time a web request takes we often operate in milliseconds ($10^{-3}s$). This means that the computer is going to very quickly go through steps 1 through 3b very quickly, then idles until a response is given and the program can continue on.

4.1.2 Some Improvements

To solve this problem we will use something called asynchronous programming. How this works we will have a response queue, R, where every time we receive a response we enqueue the response onto R. Now when we make a request to a URL instead of waiting for the response we call an *empty queue* function where we dequeue each element of R until it is empty, processing each one by finding its links and then adding the links to S and enqueuing them to Q. Once we have emptied R, then we move onto the next element of Q as we had before.[4]

The advantage of this new algorithm is that we no longer wait for the Internet's response to our requests, rather we just keep crawling and deal with responses as they get back to us. It is worth noting that there is some overhead cost incurred by this, however it will be insignificant compared to the cost of waiting for a response. There however is again an issue with this approach, scalability. While this algorithm is fast, the Internet is still magnitudes of orders larger then what this would be able to handle. If we want to make the program go faster, without changing it we have to keep investing in better and more expensive CPUs, and there is a limit to how far we can do that. Enter parallel programming.

4.1.3 Scaling Up with Parallel Programming

There are a few types of parallel programming. The differences all boil down to how expensive it is to communicate between processes; for example two threads running on different cores of the same CPU communicate very quickly as they share memory whereas programs running on hundreds of computers across the world will find it much slower to communicate with each other. In this paper we specifically deal with parallel computing where communication is costly, as this allows for the most scalability.

The most obvious way to parallelize this program is to simply run this program on many computers, each with a different seed set and sending the collected information to some central database. The issue however arises that the crawlers are going to be overlapping each other, so the benefit we get from running this on a greater number of processes is very small. To formalize the issue, every time a process goes to a page that another process has already gone to it creates a redundancy, which we consider a waste of resources.

There are three basic ways of reducing redundancy in our crawl.

- Have each process keep and updated list of what pages have already been visited by every other process. To optimize for the significant cost of communication, we may employ a buffer so that each process stores what it has crawled and only sends a message to each other process for them to update their list under some predetermined condition. This of course means that more redundancy could happen but the cost of communication is great enough that buffering becomes necessary.

- Have a master process that tells all the crawler processes what to crawl. As with the last approach, buffering is important, however here the buffering does not hurt us. The master simply send each crawler a list of URLs to crawl, once they are done they send back their results, and the master sends them a new batch. This method has benefit that it does not incur any redundancy as the master can

[4]Implementing asynchronous programming is not a trivial problem, however many programming languages have built in or standardized support for asynchronous operations making this a fairly effective tool.

ensure this. The potential downside to this method however is scalability. If there are enough crawlers running the master may be overworked and not able to keep up with them. It may be possible to unload more of the master's work onto the crawlers, however eventually one would want to find a way to distribute the master's work. This however is speculative and a single master may in-fact be enough as it can handle a tremendous amount of crawlers.

- Find a way to divide the Internet up into chunks that each crawler knows to stay within. Consider that we have a way to partition the Internet into n subsets. We may give each of our n processes a single one of these subsets to crawl. There are a couple ways to do this, for example the most simple is maybe each processes gets a specific character and if that character and it's job is to discover everything with that character being first in the URL. This is simply an example idea, however it closely relates to one of the three methods of partitioning the Internet we will consider.

  - **Hash function** - An issue with using the first letter of the URL is that not all letters are equally likely (for example `w` appears quite often at the beggining of a URL with `www`). To solve this we instead introduce the idea of a Hash function. A Hash function is a function $H : \mathbb{N} \to [n]$[5] for some $n \in \mathbb{N}$ with the requirement that given any random inputs to H the results should be uniformly distributed. An example that fits this definition is using a modulo, and much better and more complex hash functions exist, we will not cover them in this paper, rather we will simply assume their existance.

    Now that we have a hash function we may take the hash of any URL[6] and based on that determine which process should crawl that page. This ensures that each process has an equal chunk of the Internet to crawl but at the cost that a crawler will be finding many links outside it's partition.
  - **Graph structure (communities)** - Here we try and split the Internet up in a way that makes sense given the structure of it's graph. We want to try and break the Internet into communities and give each community to a different process. This becomes a difficult problem given that we don't already have the entire graph discovered. We aren't going to be going deep into this idea, however it does warrant further investigation and I will give two possible ways trying to go about this.

    * Use the already discovered parts of the graph to try and guess what community a URL might belong to. There may be some machine learning involved in this and you may need a master process or some way to coordinate between processes so that they may use even more information to inform their guesses. This process would allow for some redundancy to happen as guesses will not always be correct. That being said the amount of communication needed in this will be very minimal.
    * Use the fact that pages have content and that content relates to the graph structure of the Internet. The idea here is that we should suspect that two sites dealing with the same topic will be closer on the graph than two topics that are unrelated. We try here to levy the actual content of the pages to try and guess what the content of a link might be, and based on this we can choose to crawl it or not.

    If we were to actually implement this method, we would probably want some combination of both these methods, as they both are fundamentally trying to break the Internet graph into communities and guess what community a URL belongs to.
  - **Geographic Location** - We won't spend much time talking about this approach. The idea is simple, on the Internet messages must travel and websites are hosted somewhere. The idea is that we try and optimize response time by having crawlers all around the world dealing with websites in their geographic location. This isn't greatly scalable however and would be used in conjunction with another method.

For our purposes we are going to use the master process method as it will be easier to adjust for what is needed. Now we have a fairly scalable method of running an efficient crawling algorithm. So what is next?

[5] $[n]$ is the set $\{x \in \mathbb{N} \mid x \leq n\}$.

[6] In a computer everything may be treated as an integer if desired, even strings like URLs, so we can in fact take the Hash of a URL.

4.1.4 Age and Freshness

The Internet is not a static entity. It is always changing and we need to have our crawler reflect this. If we were able to finish a crawl of the Internet in a month, then we could keep re-crawling every month. This might be okay for most resources, but imagine someone today entering in the search query "`Trump`" Into Google and only getting results that were at least a couple weeks old, obviously we need some way to recrawl important parts quickly.

The first step in doing this is instead of our master process simply queuing everything that is found, it will use a priority queue and will have to have some protocol for adding in old pages that we want to have re-crawled. This is where Age and Freshness come in to play.

Age is the amount of time since a page has been crawled. **Freshness** is weather or not a page has been changed since it was last crawled. Both terms are widely used when talking about web crawling, however for our purposes we will only be dealing with age. We also can leverage the fact that different pages will have different requirements for how old they should be.

What we want to come up with is a priority function. This should take into account both age, content of a page some measure of importance, and possibly some other information such as how often this page has been updated in the past. We can calculate an importance by using something like PageRank, however we will not be covering any such algorithm in this paper. Once we have a priority function we can consider every page that we have discovered and already crawled to be included within our queue on the master process. When a crawler requests a new batch of pages to crawl, the master simply sends it the n URLs with the highest priority.

This finally gives us an algorithm that is fast, largely scalable, and will deal with the changing Internet. The next step is what should we do with all this data we've been discovering. It's important to realize that we aren't actually discovering the entire Internet, many pages have no links going to them. There are some ways of dealing with even this.

4.1.5 Utilizing Oddities of the Internet

Now there are some problems that come up when you are doing a massive Internet crawl. Many websites may get a lot of traffic (requests) from crawlers quickly going through all their pages, and this can slow down websites and be unintentionally harmful.[7] To deal with this, often a website will provide a `robots.txt` file that tells crawlers what they should and should not crawl. Follwing what this file tells you is purely up to you as a polite crawler.

These `robots.txt` files often can be helpful in finding new webpages on a website, that may be otherwise impossible to find. Even more helpful however is something called a sitemap. A sitemap is a file that can more or less give you all the pages and links within a website, which makes your job quite easy. You may still want to validate that the sitemap is not lieing to you, but even so it can help to discover even more potentially impossible to find parts of the web.

The last idea on finding URLs that aren't linked to is by using some educated guessing based on how websites tend to be structured. For example consider you find the URL `www.abc.com/home/index/text`, you then very reasonably may guess that `www.abc.com/home/index`, `www.abc.com/home`, and `www.abc.com` are all good candidates for possible pages.

5 Searching

Now that we have discovered our graph, or at least enough of it, we need some way to query. This is a problem that isn't well generalized. Rather then talk about how one might do this for any abstract data we will look at how we can do it on the Internet, and this will be modifiable to be usable in many situations.

The vast majority of the complexity for this comes from understanding natural language and dealing with tremendously large amounts of data, neither of which we are going to be going into any depth on, as such this section is quite short.

[7] This is the basic idea of a DDOS (Distributed Denial Of Service) attack and can be very harmful to a website.

5.1 On the Internet

Now as we are crawling the web we are continually downloading web pages and we need some way to process these. Once a page is downloaded we need to choose how and what information from it to store. We're going to store a large index of all the 'keywords' found in each web page.

In order to search quickly we want to turn this into a reverse index where we pair each keyword with the URL it comes from. Now when a query comes in, you may pick it apart however you like, find keywords matching what was searched, the serve up those pages as the result.

The last big part of this is ordering the pages, and for this there are many good algorithms. The most famous of which, PageRank will give a good measure of how important a page is based on the graph structure of the Internet. In practice you'll want to combine this with some other data, so that less significant but more relevant pages are able to show up.[8]

6 Conclusion

Now we have all the building blocks to build a search engine. We can build a high quality web crawler and we know what to do with it's results. While there may be many details left to figure out, one should have a clear idea of what steps need to be done.

7 Further Work

The next step in this research is to actually implement and test a distributed web crawler and compare a few of the different models described here and see how they compare. So far this has been beyond my means as the sort of computing power needed to test this requires hundreds of computers running in parallel. That being said, testing such an algorithm on even one computer may give some interesting results.

8 Sources

- **A Review of Web Crawler Algorithms** - `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.661.1511&rep=rep1&type=pdf`
- **The Anatomy of a Large-Scale Hypertextual Web Search Engine** - `http://infolab.stanford.edu/pub/papers/google.pdf`
- **Graph structure in the Web** - `https://dl.dropboxusercontent.com/content_link/k2RzpEomf92MtS9iavSTlhGvfile?_download_id=38209822528424053696058001979974345459371534227055105173942882812578_notify_domain=www.dropbox.com&dl=1`
- **Parallel Crawlers** - `http://ilpubs.stanford.edu:8090/733/1/2002-9.pdf`

[8] A result's relevance is how closely it's content is related to the search query, whereas it's significance is how good of a page a result is. We may measure significance with PageRank while relevance is somewhat harder to measure.