**ABSTRACT**

Most of today's world revolves around humongous amount of data, or “Big Data”. One of several problems associated with it is searching through it effectively.

Internet is one of example of big data, though there are several algorithms developed for searching across it. I have developed another Search Engine which works on previously developed concepts, and tried to understand the core concepts behind it. In another project, these core concepts then could be modified and applied to other Big Data applications.

As I learned about Search Engine, I discovered it is mainly done in three stages: Crawling across Web, Indexing the Web and then Ranking the Index.

In the process, I learned a lot about different data structures, and innovating ways in which these could be applied.

Powers of modularity, and abstraction were visible as well.

In conclusion, I hoped that everyone who pursues big data analysis, or any kind of web applications, will find that Search Engine core concepts are not only easy, but they are fun!

**INTRODUCTION**

A Web Search Engine is designed to search across World Wide Web. A typical search engine responds to a query by user, and displays the list of links in response.  
The query entered by the user is typically a combination of keywords, which s/he wants to know about, and the output by Search Engine is a “ranked” ordered of URLs (Uniform Resource Locators) that contains these keywords. Output is displayed across several pages, in what is typically referred as SERPs (Search Engine Result Pages)

An Alternative to Search Engine is a web directory. However, web directory, is maintained by humans and suffers from several limitations. They are not updated dynamically, and are extremely dependent on human bias as to which category a web page should belong. Moreover, they are limited to a set of predefined categories, and modifying the category list is behemoth task.

In contrast, Search Engine, is largely autonomous, and capable of dynamic change as web site is added, or even as content of web pages are changed. Also, their results are based on predefined criteria, and is consistent across web.

Moreover, a good ranking algorithm can rank pages, and return exactly what the user wants.

**MOTIVATION**

Today is world of Big Data. World Wide Web is one example. More examples include field of biology (DNA sequencing), meteorology, business informatics and several others. While Big Data challenges are diverse, and include problems ranging from capture, storage, analysis etc, I like to think searching as fundamental problem.

Searching the Big Data poses its unique set of problems, while at same time, if properly implemented, it make problems of Analysis a lot easier.

I focused on Web Search Engine first, because of the extensive research already done on it. Of all big data search engines, most efficient algorithms are designed for Web Search Engine. They include PageRank algorithm, reputedly the first efficient algorithm for Web search.

By learning the techniques and challenges of Web Search Engine, it is my hope, that I can expand its concept to similar fields.

**HISTORY OF SEARCH ENGINES**

As the web grew, it started becoming difficult to keep track of web pages (certainly no one can today); and thus efforts started toward an efficient search mechanisms.

At first, web pages were archived, and searched. However instead of content, only web page names were indexed, since number of web pages per query were small, and humanly readable.

However, as we Internet continued to grow, this obviously became obfuscated.

While there was no real Search Engine for much of early to mid 90s, there were efforts to “categorize” the web. They could be general or specialized and were maintained by humans. However, explosive growth of Internet, which added pages exponentially rendered this approach useless, since it was no longer feasible to read and classify the new web-pages, let alone update the old ones, if their content changes.



The earlier search engines, which appeared, were dependent on Website Administrators who would inform the search engine if any significant change takes place in website. Soon after, there was explosive growth in this area.

In late 90s, Search Engine problem attracted brightest minds of Computer Science, as each developed their own algorithms and vied for popularity. Point to be noted that, no SE company at that point of time has any way of monetizing their user base.

It was around 2000, that Google rose to prominence, due to its PageRank Algorithm. It also found a way to monetize its success by selling the links. With the dot com burst, many of earlier SE companies failed or were acquired by bigger giants.

Today, Search Engine Market is dominated by Google, known for its minimalistic interface, and quality of Search Results. The main competitors to it are Yahoo!, Bing (by Microsoft), DuckDuckGo, etc. In China, main SE is Baidu.

Moreover, exciting new thing called computational SE is in its roots. Google has already adopted several of the tenets of it. Major computational SE today is WolframAlpha.



**WORKING OF SEARCH ENGINE**

Search Engine working can be divided in three main areas:

1. **Crawling** : A web-crawler (also called spider) starts from a corpus of seed pages, and by following links on each web-page, it travels across World Wide Web.
2. **Indexing**: It is not only necessary to crawl across the Internet, but it is equally important to store the Internet, by the keywords we encounter. This is most prominently stored in { keyword : list of URL } form.
3. **Ranking :** Returning all URLs associated with keyword as they are added in list is usually not best idea. We want our result to be sorted, and make some sense, so that first few results are usually what we want.

Each of these areas come with their challenges, limitations and algorithms. I will discuss these three areas in detail, in project.

**WEB CRAWLER**

Web Crawler (also known as Spider) crawls the entire Web. It does so by taking a seed page (or group of seed pages known as corpus) and following the links given on each of these pages.

What we are calling link, is actually hyper-link, which allows two web pages to be connected to each other. It is a defining feature of Internet. Also, this is the feature that makes Internet a graph data structure.

Knowing and exploiting graph data structure is of crucial importance. Throughout search engine build, we will refer to several of terminologies and algorithms used in graph structure. Graph data structure is explored more in Appendix A.

Problem of building a web crawler can be broken down in following parts:

1. Extracting a single Link from a page
2. Using the procedure developed above, extracting all links given in a page
3. Visiting each link of step 2, and repeat.

Of course, it is not as simple as that too, as we have to keep in mind several details, like not repeating the links; maintaining the data base as to which link is to be visited and which link has already been visited.

Even then, there are several ways of traversing the Internet graph. Two very common strategies would be Depth-First Search and Breadth-First Search.

Now, we will explore each step in detail:

**Extracting a single link from a page**

To extract a link from a page, we need to understand how hyper-links work. Hyperlinks are in short, a HTML construct that allows you to refer to an URL by a name. That name is usually highlighted be being in different color and by being underlined. This tells user of a page, that if s/he pursues the Hyperlink, they will reach at another web-page related to the word represented by link.

So, in order to extract a single link, let's say a first one, I need to know how it is represented in HTML. HTML uses anchor tag for the same purpose.

So, in order to accomplish my purpose, I will search for first anchor tag in source code of page, and then exploit it to get myself an URL.

00\_first\_link

page = #content of some web page

start\_link = page.find('<a href=')

start\_quote = page.find('"', start\_link)

end\_quote = page.find('"', start\_quote + 1)

link = page[start\_quote+1:end\_quote]

**Extracting All Links from Single Page**

However, we don't want single link, we want all links of a page. I can given some content, already find first link available to that content. So, it makes sense, that as I move forward, I change the content of web-page so that new content is previous web page minus the content till last link I found.

In other words, if ith link is found on nth line, I know that (i+1)th link will be available after nth link.

So, we just keep passing content to our first\_link procedure after some modification, and the value it returns is stored by us.

This also brings us to Lists. Lists are a way in Python to store several values associated by one identifier.

02 snippet: Lists

def get\_next\_target(s) :

start\_link = s.find('<a href=')

if start\_link == -1:

return None, 0

start\_quote = s.find('"', start\_link)

end\_quote = s.find('"', start\_quote + 1)

url = s[start\_quote+1:end\_quote]

return url, end\_quote

def get\_all\_links(page):

url\_list = []

while True:

url, end\_quote = get\_next\_target(page)

if url:

page = page[end\_quote:]

url\_list.append(url)

else:

break

return url\_list

**Web Crawler**

Our final web crawler now uses the fact that we can extract all links from a page, and visit each page, storing all the links in a list “tocrawl”. As it visits each page, it removes the link from tocrawl, and move them to crawled list.

Here, the main challenges are limiting the Memory and Time (visiting each and every page on website is both time and memory intensive); so we would like to reduce that dynamic. Also, we can decide as to whether we want to visit Last Link Added first (DFS) or first link in queue should be given priority (BFS).

We can limit the number of pages we want to crawl, or we can limit the depth of pages from a seed page. One such example is given:

def crawl\_web(seed):

tocrawl = [seed]

crawled = []

while tocrawl:

page = tocrawl.pop()

if page not in crawled:

links = get\_all\_links(get\_page(page))

crawled.append(page)

union(links, tocrawl)

return crawled

**INDEX BUILDING**

Crawling the Web pages is not enough. We must know the keywords associated with the webpage. There are several ways of associating a key word with web page.

We may look in its meta-tags, headings or any other prominent text. On the other hand, we may look at all the text, and treat each word as key associated with it. Further, we may use combination of these to give some keywords more weight than others.

I have simplified approach of using all text as key, with each word having an equal weight. This is not a realistic option though, and the frequency of key is later added in. For building an index, we need to delve beyond simple data types.

**Nested Lists**

A list within list is known as Nested List. Its format is as follows:

nested\_list = [ [ x, y, z], [a, b, c], [ …. ] ….. ]

We can extend the idea to any depth level. A basic key: url list can be defined using 3 – level deep nested list:

[ [key1, [url1, url2, url3 ...] ],

[key2, [url1, url2, url3 ...] ],

]

We can even further nest it to include key count value:

[ [k1, [

[u1, count],

[u2, count],

[u3, count]

]

] ,

[k2, [

[u1, count],

[u2, count],

[u3, count],

]

]

]

Based on these structures, we can add key: url pair, search for a key, and in fact, can build an Index.

A basic addition operation is defined as:

def add\_to\_index(index, keyword, url):

for entry in index:

if keyword == entry[0]:

# print entry[1]

for link in entry[1]:

# print link

if url == link[0]:

link[1] += 1

return

entry[1].append([[url, 0]])

return

index.append([keyword, [[url, 0]]])

**HASHING**

Problem with using nested lists is that they don't scale well. After a certain point of time, a basic lookup operation using nested loop has to iterate over entire list. An index containing millions of entry may have to potentially run a million times to do a lookup operation. When thousands of users execute multiple lookups, we want our lookup operation to scale reasonably well with increase in size.

Hashing is to rescue. We will use Python Dictionary method, which hash the values intrinsically, though, Hashing is treated in Appendix B.

**DICTIONARY**

Dictionary in Python relates Key to a value, with a structure as key: value. A value is then found using dict[key].

So, for our purposes, Dictionary structure could be defined as:

{

k1 : [ [u1, count], [u2, count], [u3, count] ]

k2 : [ [u1, count], [u2, count], [u3, count] ]

}

We may decide further instead of using nested lists inside a dictionary to use nested dictionary in itself. Though, that will not offer any practical advantage except that it will logically count as key:value pair instead of two values in list.

Nested Dictionary Structure:

{

k1 :

{

u1 : count,

u2 : count,

u3 : count

}

k2 :

{

u1 : count,

u2 : count,

u3 : count

}

}

**RANKING**

A Lookup procedure based on simple Index is not satisfactory, since it simply returns all results in order they were added (or hashed). In a system, where a keyword may appear in thousands of web pages, it poses a problem. As a user, I want best possible results to occur first, which may be perhaps the page having most occurrence of keyword, page preferred by other users most, or even most “popular” webpage. It can definitively be other factors, like how many times it update, its accuracy; or even combination of several factors.

The rise of Google was attributed to its ranking algorithm. Initially, it used PageRank Algorithm, which decides on “popularity” of web-page. This algorithm was better than anything used by its contemporaries, and so people preferred Google. PageRank Algorithm is explained in detail in Appendix C. (Today, Google use over 200 factors to rank a webpage)

I have used a simplified version of PageRank algorithm as well as have used occurrence of keyword in my Ranking.

To account for these factors, we have to do several changes yet again to our existing data structure, as well as create new data structures.

A new index may look like this now:

{

k1 :

{

url\_list :

{ u1 : {

total : count, //Keyword score

score : score,

ctotal : count, //Click Score

cscore : score,

tscore : score //Total score

}

u2 : {

total : count,

score : score,

ctotal : count,

cscore : score,

...

}

},

total : count,

links\_clicked: count

}

k2: {.....

}

}

"""

**Simplified PageRank**

Now, we also have to construct a simplified PageRank algorithm. Here, I will gloss over concepts since PageRank is explained in Appendix C.

We need to create a structure which will map the graph relation of web-pages and then using this structure, we will try to find out most popular web pages.

In general, a web page is more popular if:

1. It has several inlinks to it
2. It has inlink from a popular web page
3. The inlink is prominent. (That is a web page which is referring, should have fewer outlinks)

This is of course, a simplified version, and in practice, Google has several other factors, which decide the popularity of a page, though the core remains the same.

After we obtain our graph data structure, we can compute popularity of each web page as follows:

def compute\_ranks(graph):

d = 0.8 # damping factor

numloops = 10

ranks = {}

npages = len(graph)

for page in graph:

ranks[page] = 1.0 / npages

for i in range(0, numloops):

newranks = {}

for page in graph:

newrank = (1 - d) / npages

for node in graph:

if page in graph[node]:

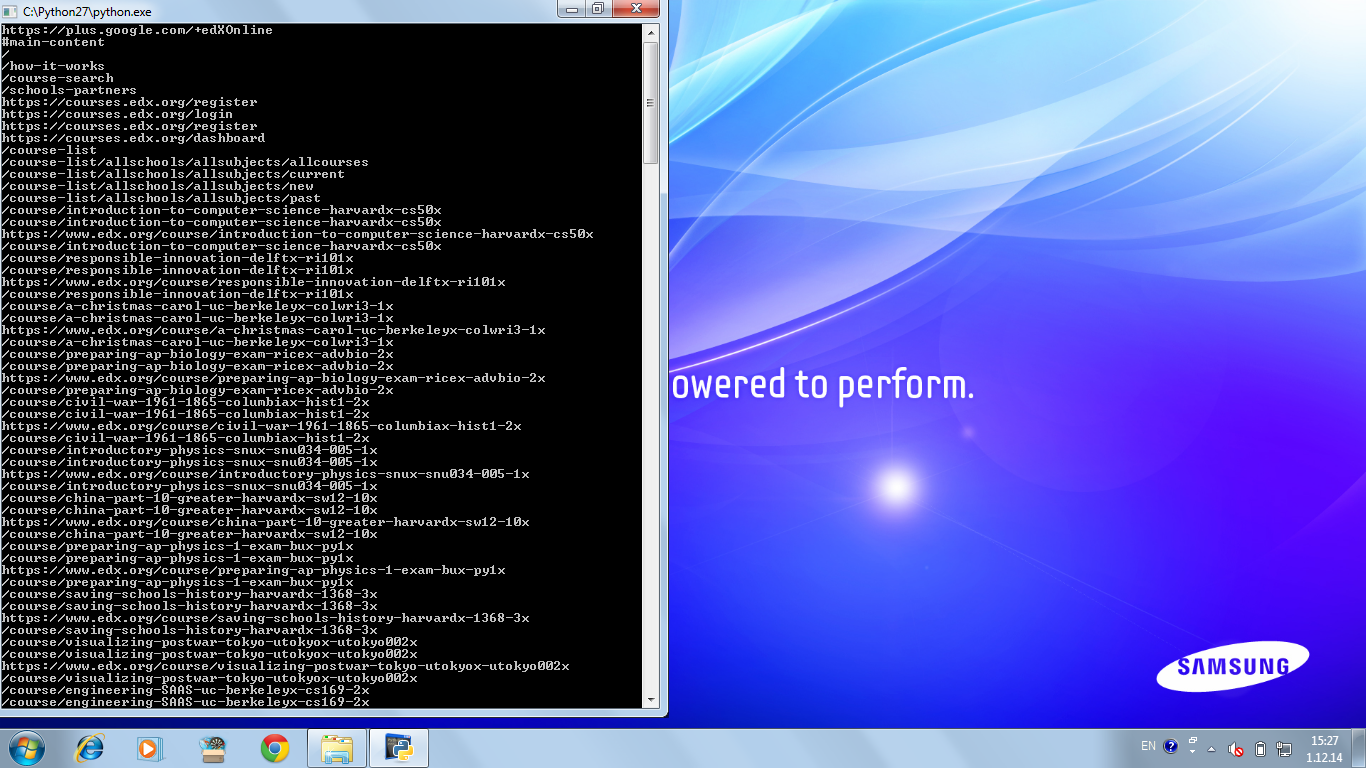
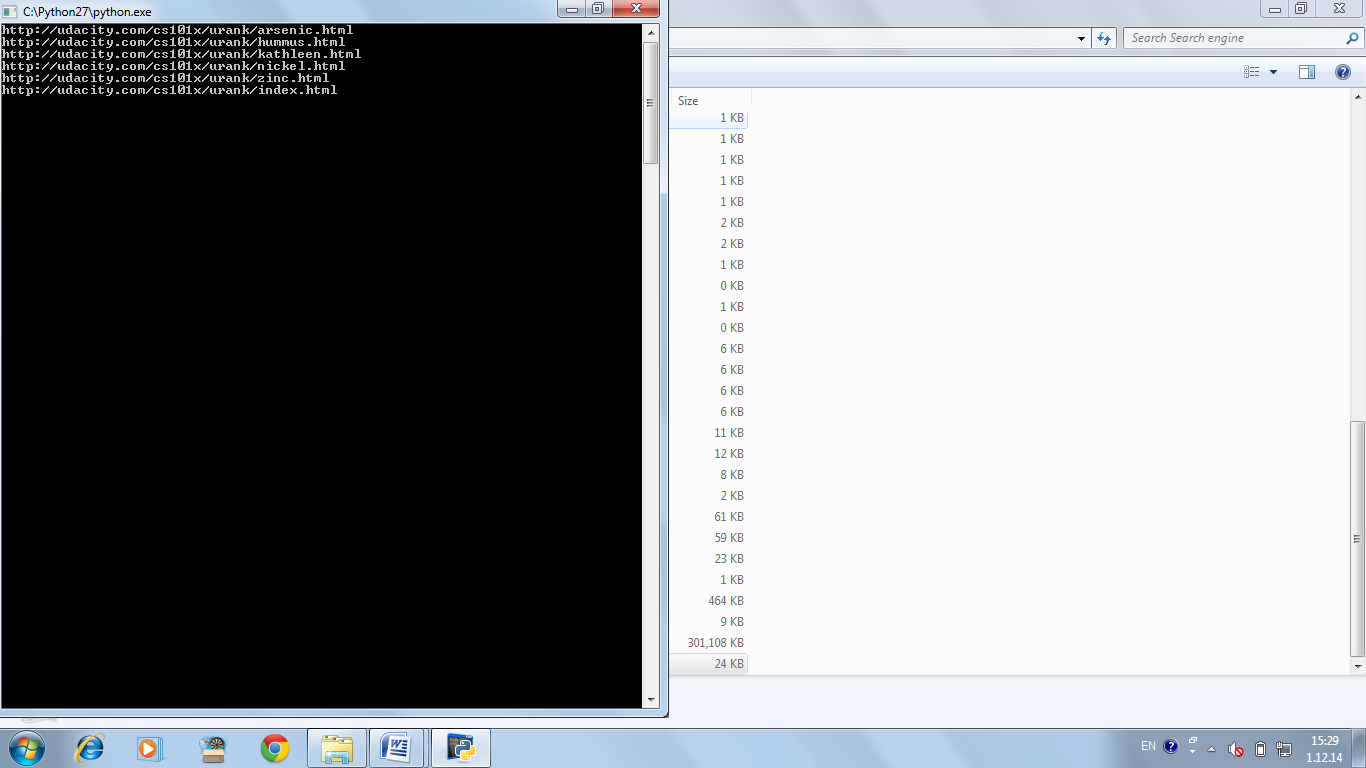
newrank = newrank + ( (d \* ranks[node]) / len(graph[node]) )

newranks[page] = newrank

ranks = newranks

return ranks

Finally, these ranks and other factors are combined, and result is returned.



**LIMITATIONS**

The Search Engine developed is basic one, and suffers from several limitations.

1. Politeness: Normally, while crawling, each web page has a “niceness” or politeness associated with it, which indicates that how many times a crawler should visit that web-page. Our crawler does not obey politeness, and thus many end up hogging up all the bandwidth of a particular server. Moreover, some websites ban crawlers who don't respect politeness, and that may very well happen with my crawler.
2. Single Machine: To build a meaningful index, we need more than one system. One system, has a practical limit both in terms of memory and time. Thus, a successful crawler is done with help of multiple machines operating simultaneously. This requires sophisticated communication between them, so that they don't step over each other toes. Our crawler is limited to single machine and its limitations.
3. Better Everything: What we have is a basic SE. It could be definitely improved in almost all aspects of SE. We can build a better crawler that runs on multiple machines, an Index, that is cloud based; and a ranking system, which takes in more factors. Apart from that, we can implement a GUI version of SE, and build a Web Application on it.

**WHY PYTHON**

I was always interested in Python as language, since it was highly recommended to me. My earlier small works in Python were full of satisfaction, due to its brevity and clear language features.

Therefore, when Udacity offered CS101: Building a Search Engine, and language accompanying it was Python, I was happy since it was my first chance to build a decent sized project in it.

Several Advantages of Python are:

1. Short Code: Code is about 2-10 times smaller than C, C++, Java
2. Maintainability : Code is extremely readable
3. Easy: Language is quite easy to learn
4. Design: Forced Indentation adds to readability

In short, this language is highly productive, in a sense, that you produce results as fast you write code. Debugging is rarely due to language quirkiness or misinterpretation of some language features.

It leaves very high amount of time to solve the problem rather than dealing with details of language.

**FUTURE**

As Exciting as Search Space is today, it is all set to become even more exciting in coming days. There are several dynamic, and explosive researches being done in Search Engine fields. Some of my personal favorites are:

1. **Personalization through History**

Instead of serving same search result to everyone, this theory argues that, everyone should see different result based on their previous search history as well as other indicators. Though privacy concerns are raised, it elevates the quality of Search Engine a several times.

1. **Semantic Search**

Instead of treating Search query as just a combination of keywords, a semantic Search Engine tries to understand the meaning of data. It uses several preprocessed tokens, as well as Natural language Processing to understand the intent of user. Semantic search systems consider various points including context of search, location, intent, variation of words, synonyms, generalized and specialized queries, concept matching and natural language queries to provide relevant search results

Facebook Open graph Search is example of Semantic Search

1. **Computational Search Engine**

Instead of traditional SERP, what if we can get direct answer to our queries? This is key concept behind computational Search Engines. They really don't index the Web, and instead rely on predefined databases. Much like Semantic Searches, they try to break down user queries, and then return answer instead of links.

WolframAlpha is currently the leading Computational Search Engine.

### APPENDIX A: GRAPH

### Graphs, vertices and edges

A *graph* is a collection of nodes called *vertices*, and the connections between them, called *edges*.

### Undirected and directed graphs

When the edges in a graph have a direction, the graph is called a *directed graph* or *digraph*, and the edges are called *directed edges* or *arcs*. Here, I shall be exclusively concerned with directed graphs, and so when I refer to an edge, I mean a directed edge. This is not a limitation, since an undirected graph can easily be implemented as a directed graph by adding edges between connected vertices in both directions.

A representation can often be simplified if it is only being used for undirected graphs, and I'll mention in passing how this can be achieved.

### Neighbours and adjacency

A vertex that is the end-point of an edge is called a *neighbour* of the vertex that is its starting-point. The first vertex is said to be *adjacent* to the second.

### Mathematical definition

More formally, a graph is an ordered pair, G = <V, A>, where V is the set of vertices, and A, the set of arcs, is itself a set of ordered pairs of vertices.

For example, the following expressions describe the graph shown above in set-theoretic language:

V = {A, B, C, D, E}

A = {<A, B>, <A, D>, <B, C>, <C, B>, <D, A>, <D, C>, <D, E>}

**TRAVERSAL**

**Breadth-First**

In graph theory, **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

**procedure** BFS(*Graph*,*source*):

create a queue *Q*

enqueue *source* onto *Q*

mark *source*

**while** *Q* is not empty:

dequeue an item from *Q* into *v*

**for each** edge *e* incident on *v* in *Graph*:

let *w* be the other end of *e*

**if** *w* is not marked:

mark *w*

enqueue *w* onto

**Depth-First**

**Depth-first search** (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

**procedure** DFS(*G*,*v*):

label *v* as discovered

**for all** edges *e* **in** *G*.adjacentEdges(*v*) **do**

**if** edge *e* is unexplored **then**

*w* ← *G*.adjacentVertex(*v*,*e*)

**if** vertex *w* is unexplored **then**

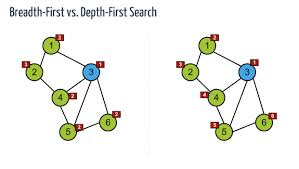
label *e* as a discovered edge

recursively call DFS(*G*,*w*)

**else**

label *e* as a back edge

label *v* as explored



**APPENDIX B : HASH**

### Mapping functions

The direct address approach requires that the function, h(k), is a one-to-one mapping from each k to integers in (1,m). Such a function is known as a perfect hashing function: it maps each key to a distinct integer within some manageable range and enables us to trivially build an O(1) search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a hash function, h(k), which maps *most* of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of collisions (cases where multiple keys map onto the same integer), is sufficiently small, then *hash tables* work quite well and give O(1) search times.

#### Handling the collisions

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

1. chaining,
2. overflow areas,
3. re-hashing,
4. using neighboring slots (linear probing),
5. quadratic probing,
6. random probing

**APPENDIX C : PAGE RANK**

**The PageRank Concept**

Since the early stages of the world wide web, search engines have developed different methods to rank web pages. Until today, the occurence of a search phrase within a document is one major factor within ranking techniques of virtually any search engine. The occurence of a search phrase can thereby be weighted by the length of a document (ranking by keyword density) or by its accentuation within a document by HTML tags.

For the purpose of better search results and especially to make search engines resistant against automatically generated web pages based upon the analysis of content specific ranking criteria (doorway pages), the concept of link popularity was developed. Following this concept, the number of inbound links for a document measures its general importance. Hence, a web page is generally more important, if many other web pages link to it. The concept of link popularity often avoids good rankings for pages which are only created to deceive search engines and which don't have any significance within the web, but numerous webmasters elude it by creating masses of inbound links for doorway pages from just as insignificant other web pages.

Contrary to the concept of link popularity, PageRank is not simply based upon the total number of inbound links. The basic approach of PageRank is that a document is in fact considered the more important the more other documents link to it, but those inbound links do not count equally. First of all, a document ranks high in terms of PageRank, if other high ranking documents link to it.

So, within the PageRank concept, the rank of a document is given by the rank of those documents which link to it. Their rank again is given by the rank of documents which link to them. Hence, the PageRank of a document is always determined recursively by the PageRank of other documents. Since - even if marginal and via many links - the rank of any document influences the rank of any other, PageRank is, in the end, based on the linking structure of the whole web. Although this approach seems to be very broad and complex, Page and Brin were able to put it into practice by a relatively trivial algorithm.

**The PageRank Algorithm**

The original PageRank algorithm was described by Lawrence Page and Sergey Brin in several publications. It is given by

PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

where

|  |  |
| --- | --- |
|  | PR(A) is the PageRank of page A, |
|  | PR(Ti) is the PageRank of pages Ti which link to page A, |
|  | C(Ti) is the number of outbound links on page Ti and |
|  | d is a damping factor which can be set between 0 and 1. |

So, first of all, we see that PageRank does not rank web sites as a whole, but is determined for each page individually. Further, the PageRank of page A is recursively defined by the PageRanks of those pages which link to page A.

The PageRank of pages Ti which link to page A does not influence the PageRank of page A uniformly. Within the PageRank algorithm, the PageRank of a page T is always weighted by the number of outbound links C(T) on page T. This means that the more outbound links a page T has, the less will page A benefit from a link to it on page T.

The weighted PageRank of pages Ti is then added up. The outcome of this is that an additional inbound link for page A will always increase page A's PageRank.

Finally, the sum of the weighted PageRanks of all pages Ti is multiplied with a damping factor d which can be set between 0 and 1. Thereby, the extend of PageRank benefit for a page by another page linking to it is reduced.

**BIBLIOGRAPHY**

1. PageRank : <http://searchengineland.com/what-is-google-pagerank-a-guide-for-searchers-webmasters-11068>
2. SE History: <http://www.searchenginehistory.com/>
3. SE History: <http://www.wordstream.com/articles/internet-search-engines-history>
4. Python Docs : <http://docs.python.org/2.7/>
5. Why Python: <http://www.python.org/doc/essays/ppt/acm-ws/sld011.htm>
6. Build a SE: <https://www.udacity.com/course/cs101>
7. Computation SE: <http://computer.howstuffworks.com/internet/basics/wolfram-alpha1.htm>
8. Graph : <http://www.martinbroadhurst.com/graph-data-structures.html>