ACKNOWLEDGEMENT

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

I am highly indebted to **Dr Naokant Deo** for his guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards my parents for their kind co-operation and encouragement which help me in completion of this project.

My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Author

Varun Hasija

2K12/MC/088

TABLE OF CONTENTS

DECLARATION BY AUTHOR

This is to declare that this project report entitled "**Search Engine for Programmers (SEP)** " submitted in the fulfillment of the course unit for the award of the degree of Bachelor of Technology in Mathematics and Computing to **Department of Applied Mathematics,** Delhi Technological University, Delhi – 110042 has been written by me. No part of the report has been plagiarized from other sources. All information from other sources has been duly acknowledged. I declare that if any part of this report is found to be plagiarized, I will take full responsibility for it.

Signature of Author

Varun Hasija

2K12/MC/088

SEARCH ENGINE FOR PROGRAMMERS
"SEP"

MINOR SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE IN BACHELOR OF TECHNOLOGY

(2012-2016)

SUBMITTED BY:

VARUN HASIJA
2K12/MC/088

Under the guidance of
DR NAOKANT DEO

DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY,DELHI

BONAFIDE CERTIFICATE

This is to certify that this project report entitled "**Search Engine for Programmers (SEP)**" submitted in partial fulfillment for the award of the degree of Bachelor of Technology in Mathematics and Computing, to **Department of Applied Mathematics,** Delhi Technological University, Delhi – 110042 is a bonafide record of project work carried out by **Varun Hasija (2K12/MC/088)** under my supervision during the 5th semester (1 August 2014 to 10 November 2014).

Signature of supervisor

DR NAOKANT DEO
DEPARTMENT OF APPLIED MATHEMATICS

ABSTRACT

Search engines allow people to search a large set of documents for a list of words/images, and get results in form of documents which are ranked according to how relevant the documents are to those words. This project aims at developing a Search engine specific to programmers for quick access of relevant and important data.

Current search engine systems face certain pitfalls:

1. Irrelevant results for programmers

   If I search for python in any given search engine, it might give results about the "snake" rather than the "programming language".

2. Unnecessary indexing of irrelevant data.

It can be said that there does not exist any search engine which would efficiently find the data from a programmer's point of view. Even if it exists, it would take a large amount of time and data to scroll through all the pages.

My project proposes and presents a model which could handle these problems efficiently.

1. INTRODUCTION

1.1 Search Engines

A **search engine** or **search service** is a program designed to help find information stored on a computer system such as the World Wide Web, inside a corporate or proprietary network or a personal computer. The search engine allows one to ask for content meeting specific criteria (typically those containing a given word or phrase) and retrieves a list of references that match those criteria. Hence, search engines allow people to search a large set of documents for a list of words/images, and get results in form of documents which are ranked according to how relevant the documents are to those words.

Search engines use regularly updated indexes to operate quickly and efficiently. *Search Engine* usually refers to a ***Web* search engine**, which searches for information on the public Web. However there are other kinds of search engines like ***enterprise search engines***, which search on intranets, ***personal search engines***, which search individual personal computers, and ***mobile search engines***. Some search engines also mine data available in newsgroups, large databases, or open directories like DMOZ.org. Unlike **Web directories**, which are maintained by human editors, search engines operate algorithmically. Most web sites which call themselves search engines are actually front ends to search engines owned by other companies.

Hence, a web search engine is a software system that is designed to search for information on the World Wide Web. The search results are generally presented in a line of results often referred to as search engine results pages (SERPs).

1.2 Information Retrieval

Information retrieval is the activity of obtaining information resources relevant to an information need from a collection of information resources. Searches can be based on metadata or on full-text (or other content-based) indexing.
Information retrieval is a huge field with a long history and search engines constitute a very small part of it.
IR can be inaccurate as long as the error is insignificant. Data is usually natural language text, which is not always well structured and could be semantically ambiguous.

1.2.1 Goal of information retrieval

The goal of a search engines is to retrieve all the documents which are relevant to a query while retrieving as few non-relevant documents as possible.

1.3 Types of Search Engines

Although the term "search engine" is often used indiscriminately to describe crawler-based search engines, human-powered directories, and everything in between, they are not all the same. Each type of "search engine" gathers and ranks listings in radically different ways.

1. Crawler-Based

Crawler-based search engines such as Google, compile their listings automatically. They "crawl" or "spider" the web, and people search through their listings. These listings are what make up the search engine's index or catalog. You can think of the index as a massive electronic filing cabinet containing a copy of every web page the Spider find. Because spiders scour the web on a regular basis, any changes you make to a web site may affect your search engine ranking.

It is also important to remember that it may take a while for a spidered page to be added to the index. Until that happens, it is not available to those searching with the search engine.

2. Directories

Directories such as Open Directory depend on human editors to compile their listings. Webmasters submit an address, title, and a brief description of their site, and then editors review the submission. Unless you sign up for a paid inclusion program, it may take months for your web site to be reviewed. Even then, there's no guarantee that your web site will be accepted.

After a web site makes it into a directory however, it is generally very difficult to change its search engine ranking. So before you submit to a directory, spend some time working on your titles and descriptions. Moreover, make sure your pages have solid well-written content.

3. Mixed Results /hybrid search engine

Some search engines offer both crawler-based results and human compiled listings. These hybrid search engines will typically favor one type of listing over the other however. Yahoo for example, usually displays human-powered listings. However, since it draws secondary results from Google, it may also display crawler-based results for more obscure queries. Many search engines today combine a spider engine with a directory service. The directory normally contains pages that have already been reviewed and accessed.

4. Pay Per Click

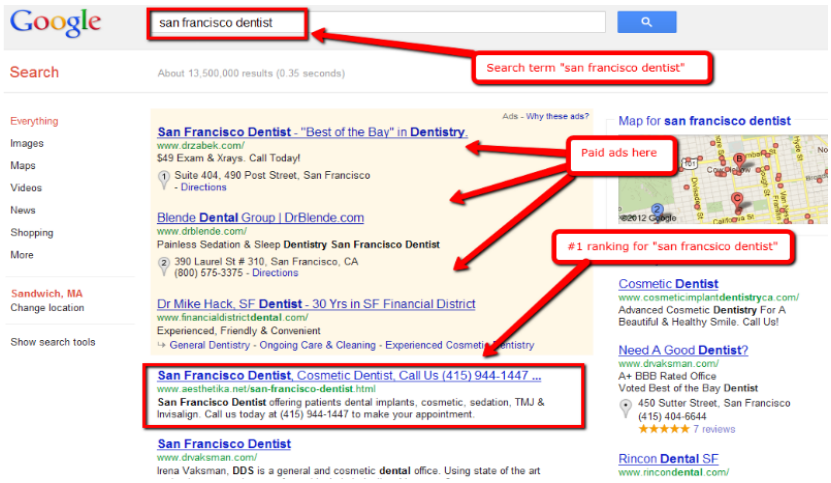
More recently, search engines have been offering very cost effective programs to ensure that your ads appear when a visitor enters in one of your keywords. This new trend is to charge you on a Cost per Click model (CPC).The listings are comprised entirely of advertisers who have paid to be there. With services such as Yahoo SM, Google AdWords, and FindWhat, bids determine search engine ranking. To get top ranking, an advertiser just has to outbid the competition.

5. Metacrawlers Or Meta Search Engines

Metasearch Engines search, accumulate and screen the results of multiple Primary Search Engines (i.e. they are search engines that search search engines) Unlike search engines, metacrawlers don't crawl the web themselves to build listings. Instead, they allow searches to be sent to several search engines all at once. The results are then blended together onto one page.

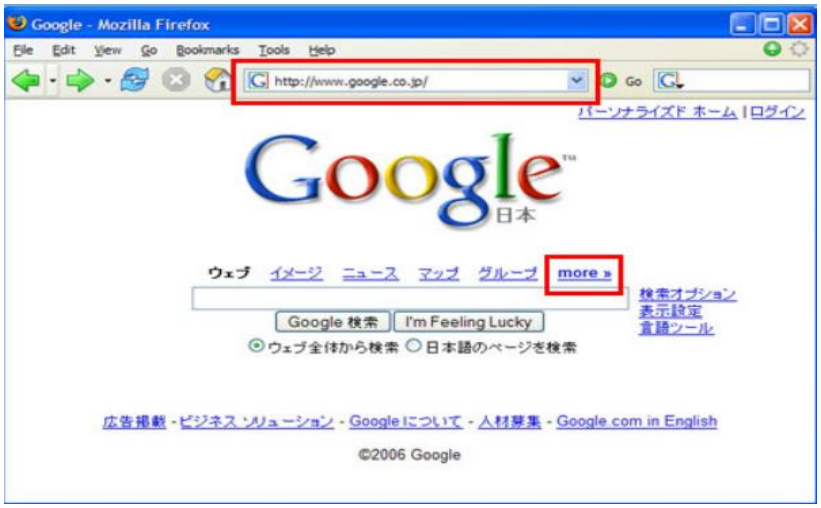
The search engine can be categorized as follows based on the application for which they are used:

1. Search by keywords

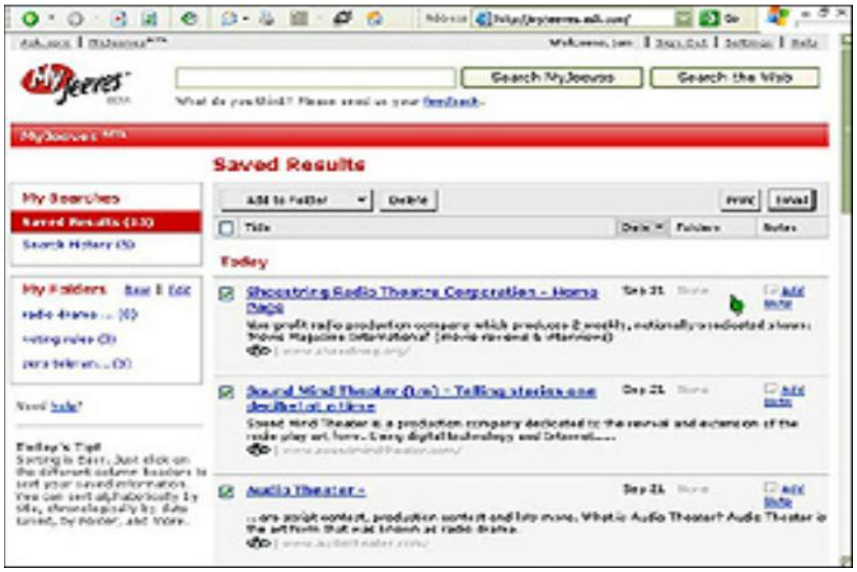


2. Search by categories

3. Search by language



4. Interview simulation



2. LITERATURE REVIEW

2.1 What's inside a search engine?

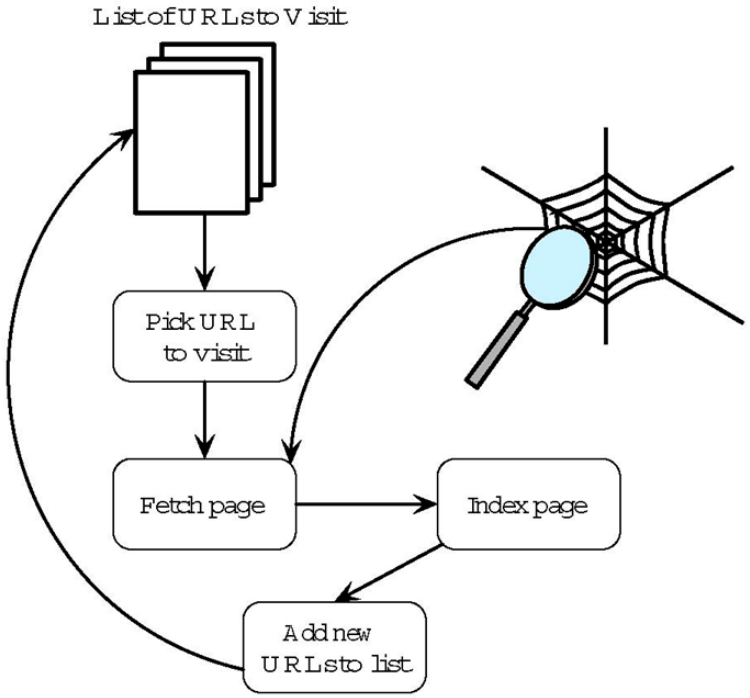
Search engines use automated software programs known as **spiders** or **bots** to survey the Web and build their databases. Web documents are retrieved by these programs and analyzed. Data collected from each web page are then added to the search engine index. When you enter a query at a search engine site, your input is checked against the search engine's index of all the web pages it has analyzed. The best URLs are then returned to you as hits, ranked in order with the best results at the top.

The working of a search engine can be thus divided into three steps

1. Web Crawling
2. Indexing
3. Ranking

2.1.1 Web Crawling

A **web crawler** (also known as a **web spider** or **web robot**) is a program or automated script which browses the World Wide Web in a methodical, automated manner. Other less frequently used names for web crawlers are **ants**, **automatic indexers**, **bots**, and **worms**



This process is called ***web crawling* or *spidering***. Many legitimate sites, in particular search engines, use spidering as a means of providing up-to-date data. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine, that will index the downloaded pages to provide fast searches. Crawlers can also be used for automating maintenance tasks on a web site, such as checking links or validating HTML code. Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting e-mail addresses (usually for spam).

A web crawler is one type of bot, or software agent. In general, it starts with a list of URLs to visit, called the **seeds**. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the **crawl frontier**. URLs from the frontier are recursively visited according to a set of policies.

2.1.2 Indexing

The web pages searched by the Web Crawlers have to be indexed for future use so that whenever search for related articles is asked the result can be provided by looking up the index. The index stores the URL along with the keywords relevant to the page. It may also store information like the date on which the page was last visited and also the date on which the page should be revisited under the revisiting policy along with the age and freshness factors.

Search engine indexing collects, parses, and stores data to facilitate fast and accurate information retrieval. The purpose of storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power.

2.1.3 Querying

Querying belongs to the search part of the search engine. A **web search query** is a query that a user enters into a web search engine to satisfy his or her information needs. Web search queries are distinctive in that they are often plain text or hypertext with optional search-directives (such as "and"/"or" with "-" to exclude). They vary greatly from standard query languages, which are governed by strict syntax rules as command languages with keyword or positional parameters.
Query results in the display of a list of pages which are ranked based on the relevance to different users.

th

2.1.4 Ranking models

Most of the search engines return results with confidence or relevancy rankings. In other words, they list the hits according to how closely they think the results match the query. However, these lists often leave users shaking their heads on confusion, since, to the user, the results may seem completely irrelevant.

Basically this happens because search engine technology has not yet reached the point where humans and computers understand each other well enough to communicate clearly.

Most search engines use search term frequency as a primary way of determining whether a document is relevant. If you're researching diabetes and the word "diabetes" appears multiple times in a Web document, it's reasonable to assume that the document will contain useful information. Therefore, a document that repeats the word "diabetes" over and over is likely to turn up near the top of your list.

If your keyword is a common one, or if it has multiple other meanings, you could end up with a lot of irrelevant hits. And if your keyword is a subject about which you desire information, you don't need to see it repeated over and over--it's the information *about* that word that you're interested in, not the word itself.

Some search engines consider both the frequency and the positioning of keywords to determine relevancy, reasoning that if the keywords appear early in the document, or in the headers, this increases the likelihood that the document is on target. For example, one method is to rank hits according to how many times your keywords appear and in which fields they appear (i.e., in headers, titles or plain text). Another method is to determine which documents are most frequently linked to other documents on the Web. The reasoning here is that if other folks consider certain pages important, you should, too.

As far as the user is concerned, relevancy ranking is critical, and becomes more so as the sheer volume of information on the Web grows. Most of us don't have the time to sift through scores of hits to determine which hyperlinks we should actually explore. The more clearly relevant the results are, the more we're likely to value the search engine.

We can provide scores to the different pages for a given query based on:

1. Content based ranking
2. Inbound link ranking

2.1.4.1 Content based ranking

There are several ways to calculate a score based only on the query and the content of the page. These scoring metrics include:

Word frequency
The number of times the words in the query appear in the document can help determine how relevant the document is.

Document location
The main subject of a document will probably appear near the beginning of the document.

Word distance
If there are multiple words in the query, they should appear close together in the document.

2.1.4.2 Using Inbound links

Although many search engines still works in one of the above explained ways, the results can often be improved by considering information that others have provided about the page, specifically, who has linked to the page and what they have said about it. This is particularly useful when indexing pages of dubious value or pages that might have been created by spammers, as these are less likely to be linked than pages with real content.

Simple Count
The easiest thing to do with inbound links is to count them on each page and use the total number of links as a metric for the page. Academic papers are often rated in this way.

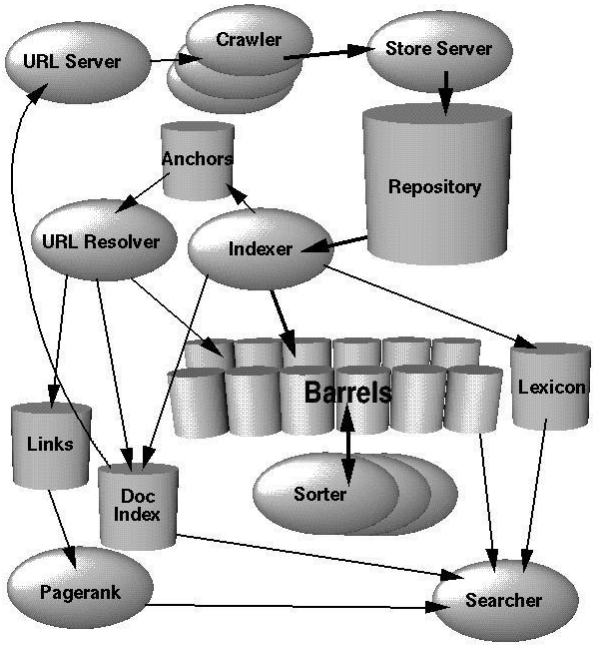
PageRank Algorithm
In theory, PageRank (named after one of its inventors, Larry Page) calculates the probability that someone randomly clicking on links will arrive at a certain page. The more inbound links the page has from other popular pages, the more likely it is that someone will end up there purely by chance. Of course, if the user keeps clicking forever, they'll eventually reach every page, but most people stop surfing after a while. To capture this, PageRank also uses a *damping factor* of 0.85, indicating that there is an 85 percent chance that a user will continue clicking on links at each page.

2.2 Google Search Engine

GOOGLE *is one of the popular and most widely used and efficient search engines. Hence, here we discuss the various policies used by Google in web crawling, indexing and ranking.*

2.2.1 Google's Architecture

Most of Google is implemented in C or C++ for efficiency and can run in either Solaris or Linux.



Crawlers: In Google, the web crawling (downloading of web pages) is done by several distributed crawlers.

URL Servers: The URLserver sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the **storeserver.**

Store Server: The storeserver compresses and stores the web pages into a repository. Every web page has an associated ID number called a **docID** which is assigned whenever a new URL is parsed out of a web page.

Indexer: The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of **"barrels"**, creating a partially sorted forward index.

URL Resolver: The URL Resolver reads the Anchor files and in turn converts relative URL into absolute URL and then into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

Sorter: The sorter takes the barrels, which are sorted by docID and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index.

Lexicon: A program called **DumpLexicon** takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

3 PROBLEM STATEMENT

Till now, we have seen how a typical search engine like Google would work.

Limitations of existing search engines from programmer's point of view:

1. Too many generic results for a given keyword

For ex. when we search for python on Google it might return results about "snake" instead of the "programming language".

2. Unnecessary indexing of irrelevant data
Data such as a musician with python is irrelevant for programmers.

It can be said that there does not exist any search engine which would efficiently find the data from a programmer's point of view. Even if it exists, it would take a large amount of time and data to scroll through all the pages.

This project aims at developing a Search engine specific to programmers for quick access of relevant and important data.

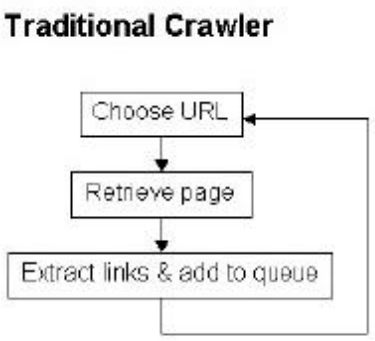
4 PROPOSED MODEL

As we know, a search engine can be divided in 3 parts:

1. Web crawler
2. Indexing
3. Searching

We explain the working of back-end of search engine written in python and then explain the Graphic User Interface developed using Tkinter library in python.

4.1 Web Crawler



A web crawler is one type of bot, or software agent. In general, it starts with a list of URLs to visit, called the **seeds**. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the **crawl frontier**. URLs from the frontier are recursively visited according to a set of policies.

We select the page http://kiwitobes.com/wiki as a seed and feed it into the crawler function.

Programming language used: Python

API used:

urllib2 - to download the pages
BeautifulSoup - for parsing the webpage and building a structured representation

Mode of crawl:
BFS

Now, I present the code for crawler explained inline.

```
1.  import urllib2
2.  from BeautifulSoup import *
3.  from urlparse import urljoin
4.
5.  # Create a list of words to ignore
6.  ignorewords={'the':1,'of':1,'to':1,'and':1,'a':1,'in':1,'is':1,'it':1}
7.
8.
9.  class crawler:
10.   # Initialize the crawler with the name of database
11.   def __init__(self,dbname):
12.     pass
13.   def __del__(self):
14.     pass
15.   def dbcommit(self):
16.     pass
17.
18. # Auxilliary function for getting an entry id and adding
19. # it if it's not present
20.   def getentryid(self,table,field,value,createnew=True):
21.     return None
22. # Index an individual page
23.   def addtoindex(self,url,soup):
24.     print 'Indexing %s' % url
25. # Extract the text from an HTML page (no tags)
26.   def gettextonly(self,soup):
27.     return None
28. # Separate the words by any non-whitespace character
29.   def separatewords(self,text):
30.     return None
31.
32. # Return true if this url is already indexed
33.   def isindexed(self,url):
34.     return False
35. # Add a link between two pages
36.   def addlinkref(self,urlFrom,urlTo,linkText):
37.     pass
38. # Starting with a list of pages, do a breadth
39. # first search to the given depth, indexing pages
40. # as we go
41.   def crawl(self,pages,depth=2):
42.     pass
43. # Create the database tables
44.   def createindextables(self):
```

```
    pass
# Starting with a list of pages, do a breadth
  # first search to the given depth, indexing pages
  # as we go
  def crawl(self,pages,depth=2):
    for i in range(depth):
      newpages=set()
      for page in pages:
        try:
          c=urllib2.urlopen(page)
        except:
          print "Could not open %s" % page
          continue
        soup=BeautifulSoup(c.read())
        self.addtoindex(page,soup)

        links=soup('a')
        for link in links:
          if ('href' in dict(link.attrs)):
            url=urljoin(page,link['href'])
            if url.find("'")!=-1: continue
            url=url.split('#')[0]  # remove location portion
            if url[0:4]=='http' and not self.isindexed(url):
              newpages.add(url)
            linkText=self.gettextonly(link)
            self.addlinkref(page,url,linkText)

        self.dbcommit()

      pages=newpages
```

This function loops through the list of pages, calling *addtoindex* on each one (right now this does nothing except print the URL). It then uses Beautiful Soup to get all the links on that page and adds their URLs to a set called *newpages*. At the end of the loop, *newpages* becomes *pages*, and the process repeats.

This function can be defined recursively so that each link calls the function again, but doing a breadth-first search allows for easier modification of the code later, either to keep crawling continuously or to save a list of un-indexed pages for later crawling. It also avoids the risk of overflowing the stack.

4.2 Building the Index

The next step is to set up the database for the full-text index. The index is a list of all the different words, along with the documents in which they appear and their locations in the documents. Also, we'll be looking at the actual text on the page and ignoring non-text elements. We'll be indexing individual words with all the punctuation characters removed.

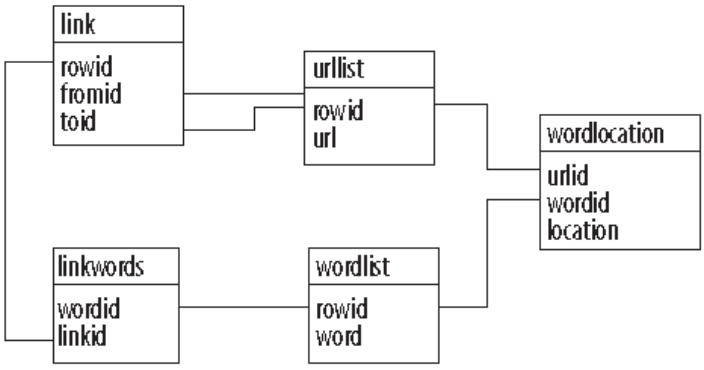
The method for separating words is not perfect, but it will suffice for building a basic search engine. Stemming algorithm can be used to perfectly separate the words according to the query.

SQLite is an embedded database that is very easy to set up and stores a whole database in one file. SQLite uses SQL for queries and hence, can be handled easily.

4.2.1 Database schema

The schema for the basic index is five tables.

The first table *(urllist)* is the list of URLs that have been indexed. The second table *(wordlist)* is the list of words, and the third table *(wordlocation)* is a list of the locations of words in the documents. The remaining two tables specify links between documents. The *link* table stores two URL IDs, indicating a link from one table to another, and *linkwords* uses the wordid and linkid columns to store which words are actually used in that link.



4.2.2 Finding and storing the words

The files that you're downloading from the Web are HTML and thus contain a lot of tags, properties, and other information that doesn't belong in the index. The first step is to extract all the parts of the page that are text. We can do this by searching the soup for text nodes and collecting all their content. A function *gettextonly* is defined for this purpose. The function returns a long string containing all the text on the page.

Next is the *separatewords* function, which splits a string into a list of separate words so that they can be added to the index.

After this, we define a method *addtoindex*. This method will call the two functions that were defined above to get a list of words on the page. Then it will add the page and all the words to the index, and will create links between them with their locations in the document.

You'll also need this to update the helper function *getentryid*. All this does is return the ID of an entry. If the entry doesn't exist, it is created and the ID is returned.

Finally, you'll need to fill in the code for *isindexed*, which determines whether the page is already in the database, and if so, whether there are any words associated with it.

The following additions are done to the above explained code:

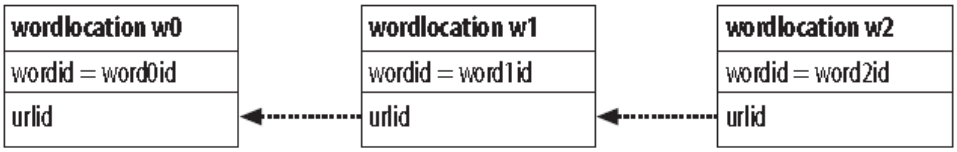
```
1.  import sqlite3
2.  from urlparse import urljoin
3.  from sqlite3 import dbapi2 as sqlite
4.
5.
6.  class crawler:
7.    # Initialize the crawler with the name of database
8.    def __init__(self,dbname):
9.      self.con=sqlite.connect(dbname)
10.   def __del__(self):
11.     self.con.close()
12.   def dbcommit(self):
13.     self.con.commit()
14.
15. # Auxilliary function for getting an entry id and adding
16. # it if it's not present
17.
18.   def getentryid(self,table,field,value,createnew=True):
```

```
    cur=self.con.execute(
    "select rowid from %s where %s='%s'" % (table,field,value))
    res=cur.fetchone( )
    if res==None:
      cur=self.con.execute(
      "insert into %s (%s) values ('%s')" % (table,field,value))
      return cur.lastrowid
    else:
      return res[0]
# Index an individual page

  def addtoindex(self,url,soup):
    if self.isindexed(url): return
    print 'Indexing '+url
    # Get the individual words
    text=self.gettextonly(soup)
    words=self.separatewords(text)
    # Get the URL id
    urlid=self.getentryid('urllist','url',url)
    # Link each word to this url
    for i in range(len(words)):
      word=words[i]
      if word in ignorewords: continue
      wordid=self.getentryid('wordlist','word',word)
      self.con.execute("insert into wordlocation(urlid,wordid,location) \
        values (%d,%d,%d)" % (urlid,wordid,i))
# Extract the text from an HTML page (no tags)

  def gettextonly(self,soup):
    v=soup.string
    if v==None:
      c=soup.contents
      resulttext=''
      for t in c:
        subtext=self.gettextonly(t)
        resulttext+=subtext+'\n'
      return resulttext
    else:
      return v.strip( )

# Separate the words by any non-whitespace character

  def separatewords(self,text):
    splitter=re.compile('\\W*')
    return [s.lower( ) for s in splitter.split(text) if s!='']

```

```
65. # Return true if this url is already indexed
66.
67.   def isindexed(self,url):
68.     u=self.con.execute("select rowid from urllist where url='%s'" % url).fetchone( )
69.     if u!=None:
70.       # Check if it has actually been crawled
71.       v=self.con.execute('select * from wordlocation where urlid=%d' % u[0]).fetchone( )
72.       if v!=None: return True
73.     return False
74.
75.
76.   def createindextables(self):
77.     self.con.execute('create table urllist(url)')
78.     self.con.execute('create table wordlist(word)')
79.     self.con.execute('create table wordlocation(urlid,wordid,location)')
80.     self.con.execute('create table link(fromid integer,toid integer)')
81.     self.con.execute('create table linkwords(wordid,linkid)')
82.     self.con.execute('create index wordidx on wordlist(word)')
83.     self.con.execute('create index urlidx on urllist(url)')
84.     self.con.execute('create index wordurlidx on wordlocation(wordid)')
85.     self.con.execute('create index urltoidx on link(toid)')
86.     self.con.execute('create index urlfromidx on link(fromid)')
87.     self.dbcommit( )
```

4.3 Querying

For querying, a new class *searcher* is defined. The *wordlocation* table gives an easy way to link words to tables, so it is quite easy to see which pages contain a single word. However, a search engine is pretty limited unless it allows multiple-word searches. To do this, we'll need a query function that takes a query string, splits it into separate words, and constructs a SQL query to find only those URLs containing all the different words. A method *getmatchrows* which creates a reference to the *wordlocation* table for each word in the list and joining them all on their URL IDs



So, a query for two words with the IDs 10 and 17 becomes:

select w0.urlid,w0.location,w1.location
from wordlocation w0,wordlocation w1
where w0.urlid=w1.urlid
and w0.wordid=10
and w1.wordid=17

The following additions are done to *searcher* class defined above.

```
1.  class searcher:
2.    def __init__(self,dbname):
3.      self.con=sqlite.connect(dbname)
4.
5.    def __del__(self):
6.      self.con.close()
7.
8.    def getmatchrows(self,q):
9.      # Strings to build the query
10.     fieldlist='w0.urlid'
11.     tablelist=''
12.     clauselist=''
13.     wordids=[]
14.     # Split the words by spaces
15.     words=q.split(' ')
16.     tablenumber=0
17.
18.     for word in words:
19.       # Get the word ID
```

```
20.       wordrow=self.con.execute(
21.       "select rowid from wordlist where word='%s'" % word).fetchone( )
22.       if wordrow!=None:
23.         wordid=wordrow[0]
24.         wordids.append(wordid)
25.         if tablenumber>0:
26.           tablelist+=','
27.           clauselist+=' and '
28.           clauselist+='w%d.urlid=w%d.urlid and ' % (tablenumber-1,tablenumber)
29.         fieldlist+=',w%d.location' % tablenumber
30.         tablelist+='wordlocation w%d' % tablenumber
31.         clauselist+='w%d.wordid=%d' % (tablenumber,wordid)
32.         tablenumber+=1
33.     # Create the query from the separate parts
34.     fullquery='select %s from %s where %s' % (fieldlist,tablelist,clauselist)
35.    # print fullquery
36.     cur=self.con.execute(fullquery)
37.     rows=[row for row in cur]
38.
39.     return rows,wordids
```

4.4 Ranking

So far we've managed to retrieve pages that match the queries, but the order in which they are returned is simply the order in which they were crawled. In a large set of pages, you would be stuck sifting through a lot of irrelevant content for any mention of each of the query terms in order to find the pages that are really related to your search. To address this issue, you need ways to give pages a *score* for a given query, as well as the ability to return them with the highest scoring results first.

For this purpose, we use the *word frequency* metric which scores a page based on how many times the words in the query on that page.

Further, a *Normalization* method is also defined which will take a dictionary of IDs and scores and return a new dictionary with the same IDs, but with scores between 0 and 1. Each score is scaled according to how close it is to the best result, which will always have a score of 1.

These functions are added to the *searcher* class defined above:

```
1.  def getscoredlist(self,rows,wordids):
2.    totalscores=dict([(row[0],0) for row in rows])
3.
4.    weights=[(1.0,self.frequencyscore(rows))]
5.    for (weight,scores) in weights:
6.      for url in totalscores:
7.        totalscores[url]+=weight*scores[url]
8.    return totalscores
9.
10.  def geturlname(self,id):
11.   return self.con.execute(
12.     "select url from urllist where rowid=%d" % id).fetchone( )[0]
13.
14.  def query(self,q):
15.   list=[]
16.   rows,wordids=self.getmatchrows(q)
17.   scores=self.getscoredlist(rows,wordids)
18.   rankedscores=sorted([(score,url) for (url,score) in scores.items( )],reverse=1)
19.   for (score,urlid) in rankedscores[0:10]:
20.     list.append(score)
21.     list.append(self.geturlname(urlid))
22.     print '%f\t%s' % (score,self.geturlname(urlid))
23.     #print '%s' % (self.geturlname(urlid))
24.   return list
25.
26.  def normalizescores(self,scores,smallIsBetter=0):
```

```
27.     vsmall=0.00001 # Avoid division by zero errors
28.     if smallIsBetter:
29.       minscore=min(scores.values( ))
30.       return dict([(u,float(minscore)/max(vsmall,l)) for (u,l) in scores.items( )])
31.     else:
32.       maxscore=max(scores.values( ))
33.       if maxscore==0: maxscore=vsmall
34.       return dict([(u,float(c)/maxscore) for (u,c) in scores.items( )])
35.
36.   def frequencyscore(self,rows):
37.     counts=dict([(row[0],0) for row in rows])
38.     for row in rows: counts[row[0]]+=1
39.     return self.normalizescores(counts)
```

4.5 Graphic User Interface

In computing, a **graphical user interface** (**GUI**, sometimes pronounced "gooey" (or "gee-you-eye")) is a type of interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation.

A proper GUI is a pre-requisite for developing a search engine. For developing the GUI, we use a library *Tkinter* in python. The following is the code for GUI which is fairly easy to be understood.

```
1.  from Tkinter import *
2.  import crawler_index_querying
3.  from PIL import ImageTk,Image
4.
5.  def loadEntry  () :
6.      e=crawler_index_querying.searcher('searchindex.db')
7.      s=e.query(nameVar.get())
8.      T.delete("1.0",END)
9.
10.     for i in s:
11.       T.insert(END,i)
12.       T.insert(END,"\n")
13.
14.
15. def makeWindow () :
16.     global nameVar, select,T
17.     win = Tk()
18.     win.title("Varun's Personal Search Engine for Programming Languages")
19.     im = Image.open('t1.png')
20.     im=im.resize((700,150),Image.ANTIALIAS)
21.     tkimage = ImageTk.PhotoImage(im)
22.     myvar=Label(win,image = tkimage)
23.     myvar.image=tkimage
24.     myvar.pack()
25.
26.     frame1 = Frame(win)
27.     l=Label(win,text="WELCOME TO MY SEARCH ENGINE",font=("comic sans ms",
        20),fg="black")
28.     l.pack()
29.     Label(frame1, text="Query",font=("comic sans ms", 16)).grid(row=0, column=0, stick
    y=W)
30.     nameVar = StringVar()
31.     name = Entry(frame1, textvariable=nameVar,bd=5,width=50)
32.     name.grid(row=0, column=1, sticky=W)
```

```
    frame1.pack(expand=True)

    frame2 = Frame(win)
    frame2.pack()
    b1 = Button(frame2,text=" Search ",command=loadEntry,font=("comic sans ms", 12))

    b1.pack(side=LEFT)

    frame3= Frame(win)
    T = Text(win, height=30, width=100, bd=5,spacing1=2)
    T.pack()
    T.delete("1.0",END)
    frame3.pack()


win = makeWindow()
mainloop()
```

This code results in the following GUI:

5 CONCLUSION AND FUTURE SCOPE

5.1 Conclusion

The main purpose of this project was to develop a simple yet efficient search engine which would return results based on what a programmer requires the most. I presented a model to tackle the general pitfalls faced by a search engine. Python was used as the both back-end and front-end programming language as it suited to be the most viable option available. GUI presented is simple, provides direct and fast results to the user. The availability of this Engine off-line also makes it a viable asset to a programmer.

5.2 Future scope

In the future, we can apply self learning algorithms to this search engine to make the results even more useful to the user. We can make the system run on-site and generate the latest data for the user. Also, we can categorize the results into images/files/docs/videos etc by further enhancing the GUI.

Thus, a lot of work is still needed to be done to fully develop the software and making it available online for use. It would require a team of developers to turn this into a big scale project.

6. REFERENCES

- http://en.wikipedia.org/wiki/Web_search_engine
- http://segaran.com/wiki
- http://www.amazon.com/gp/product/0596529325/ref=as_li_qf_sp_asin_il?ie=UTF8&camp=1789&creative=9325&creativeASIN=0596529325&linkCode=as2&tag=tasktoy-20
- https://class.coursera.org/ml-006
- http://www.it.iitb.ac.in/~cna/dokuwiki/media/ranjith/iit_report.pdf
- http://www.planetb.ca/syntax-highlight-word