Class Project Final Paper

## Project 5: Web Search Engine

**Students** (listed alphabetically by last name then first):

Sanner Barnes :: [sannerbarnes@gmail.com](mailto:sannerbarnes@gmail.com)

Leith Brandeland :: [leith98b@gmail.com](mailto:leith98b@gmail.com)

Josh Guthrie :: [josh.x.guthrie@gmail.com](mailto:josh.x.guthrie@gmail.com)

Joseph Johnson :: [joejunior1984@gmail.com](mailto:joejunior1984@gmail.com)

Revarr Johnson :: [rmj092020@utdallas.edu](mailto:rmj092020@utdallas.edu)

Revanth Segu :: [Revanth.Segu@utdallas.edu](mailto:Revanth.Segu@utdallas.edu)

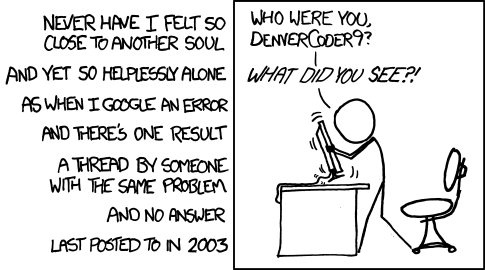
## The Problem

We’ve decided as a group to build a search engine to cater to programmers that can be used to lookup solutions to common programming problems, such as programming errors encountered. For our topics, we chose each one to be a different programming language (we have up to four programming language topics that we concentrated on: Perl, C, C++ and Java). For the most part these languages all live in the C family of languages (Perl lives in a world of its own, but it shares some of its syntax and commands with C).

Per the project requirements we could only crawl up to about 2000 links/documents, and of course this is limiting in researching programming errors, but we made every effort to crawl specialized websites from DMOZ for each language (e.g. popular help forums for each language, such as stackoverflow.com) in hopes of capturing a broader, language-specialized sample of programming problems for each of our chosen programming languages.

### *Inspiration*

When we got together to discuss choices for our search engine content, we ran through several different ideas, such as a music lyrics search engine, until we started fishing for each other’s common interests in an attempt to discover search topics that we frequently performed. It turns out, unsurprisingly I might add, that the number one search task we performed on search engines as computer science students—and as, for many of us, programming professionals in the corporate setting—was programming related search tasks; specifically, we were frequently trying to debug coding errors. This comic we found also served as inspiration for our crawling and indexing motivation:



#### Programming Paradigms

One thing we immediately recognized was that many of us, especially those in the corporate setting, were all very good at finding quick solutions to our programming problems. One of the first things we did was to capture a brief summary of the kinds of things we searched for and try and concentrate our crawl on sites that specialized in these things. It turned out that there were three main paradigms of programming that we all frequently searched for:

* **Programming Errors**:  
  [*where we have an immediate problem we are trying to solve]*
  + Sample query: Syntax Error: Line 12. IndexOutOfBoundsException.
* **API Documentation lookups**   
  [*where we want to discover arguments, see examples of usage and so forth]*.
  + Sample query: Java.lang.StringBuilder substring
* **Hello Worlds**  
  [*where we are trying to bootstrap ourselves into a new programming paradigm]*
  + Sample query: perl hello world

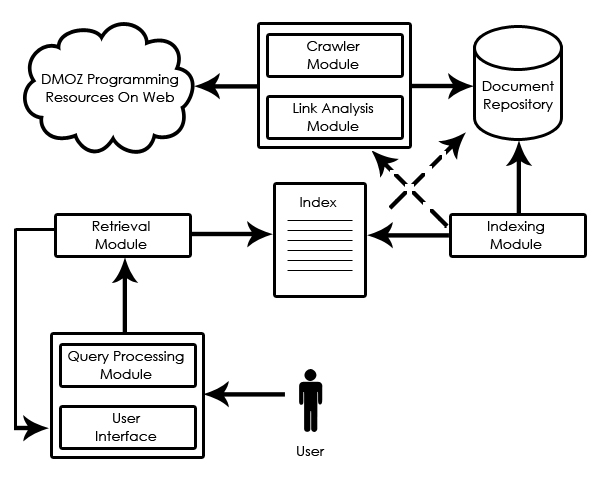
#### A brief overview of the things we did.

It was very clear from our practice in the assignments and in the material we learned in both the lectures and the IR textbook, that a basic search engine as its most basic level consisted of a *crawling mechanism* to fetch documents from a source such as the web [and subsequently parse/discover new anchor URL’s of new pages to parse], an *inverted index* of term to document/hits, a *document repository* or some kind of forward index (to display document properties after retrieval, such as showing results highlighting or document titles and a URL), a *retrieval module* for scoring and ranking results, and a search user interface to perform the search and display the results.

Additionally, beyond these basic things, a search engine might do link analysis with the URL’s it crawls (using a links db), with some form of link ranking algorithm such as HITS or PageRank. We did this in our project as well, and we will discuss these topics and more in our paper.

## The Methodology, and a block diagram of the project.

The search engine that we designed was comprised of the following features:



We explain the purpose of each major module below:

* *Crawling Module*:  
  Collects pages from our chosen sites in DMOZ [seed urls] up to the min/max document requirements for links fetched/crawled. We examined popular seed URL’s from each of our langauges on DMOZ. We also examined various crawlers and settled on Apache Nutch for this.
* *Indexing Module and Document Repository*:   
  We needed a module that would store our documents/postings as an inverted index to handle user queries. We settled on Apache SOLR for this, which uses Lucene as the indexing method. We found it integrated well with Nutch’s crawling. Lucene and SOLR also handled storing the document’s content in a Document Repository (for query results highlighting).
* *Link Analysis Module*:  
  To perform our link analysis—rewarding pages with higher links pointing to it—we used an implementation of PageRank called LinkRank, which was built into Nutch. Essentially it is equivalent to PageRank, except by default Nutch ignores outlinks from a website to itself in its scores (such that a website can’t increase its own rank simply by posting numerous links to itself—but also since our crawl was small, if we concentrated on only a few domains, we didn’t want one domain with very high outlinks to itself but very few outlinks from other sites, to rank higher against sites with legitimate outlinks in our crawl).
* *Retrieval Module*:  
  We used a combination of two retrieval modules; the first was a Boolean retrieval module, and the second was a vector space scoring module. SOLR supports both modules natively. The VSM in SOLR uses a modified tf-idf scheme that takes boosting into account (such that the linkrank scores we computed can boost the results that SOLR returns from Lucene) and that also rewards documents that contain a higher percentage of terms from the query (called the “coordination factor” by Lucene/SOLR).
* *Query Processing Module:*We wrote a custom UI for our search engine that uses AJAX to perform the searching through SOLR. Additionally we displayed pre-filled links to Google and Bing on the interface for results comparison against professional search engines. In query processing, the query is indexed and preprocessed in the same manner as documents that go into the search engine—and this is done by SOLR/Lucene. In the results we displays the best results, sorted by highest relevance/link-boosted scores first; we included the document title, URL, and a preview of the document’s content [with query term highlighting]. We also display the query term that it took to process it (similar to Google).

## Lucene Tokenization

Lucene is the back-end for SOLR. It performs the tokenization and a process known as an Analyzer that performs the following functions:

* + Case folding
  + Stop-word filtering
  + Apostrophe removal: ‘
  + Stemming
  + Punctuation removal from acronyms   
    (T.L.A becomes TLA)

Lucene hosts and creates the inverted index. It also keeps track of the anchor text for each link and serves as a document repository, together with SOLR (for query highlighting). The search is performed at runtime using these inverted and forward indices.

## Deduplication and Similarity Signatures

Our similarity detection is thanks to the deduplication abilities of Solr. Solr uses a fingerprint to detect duplicates, and provides for customization of this feature within the engine. This works by adding a new column to the document database, and storing a fingerprint there generated by one of three methods: MD5, Lookup3, and Text Profile. If the signatures for a pair of documents are the same, then the engine spots those as duplicates, and can either remove the duplicate documents or keep them depending on preferences built into Solr's configuration files.

MD5 and Lookup3 signatures look for exact duplicates by taking a hash of the contents of the document. MD5 uses a 128-bit hash, and Lookup3 uses a 64-bit hash which is just faster and easier to store than the MD5 hash. The exact portions of the document that are hashed can be set when setting up the deduplication engine.

The MD5 and Lookup3 signatures simply generate a hash of the document's plain-text content. The Text Profile, on the other hand, does a customizable similarity signature that results in a 128-bit fingerprint. Like the other signatures, the exact sections of the document that are considered part of the plain-text content can be included as part of the configuration file solrconfig.xml.

The Text Profile signature is created in a several step process. First, the plain-text document is broken into terms, and then particular terms are removed that don't meet certain length minimums. Then those terms that remain have their frequency reduced to the nearest multiple of the QUANT variable, which is the product of a user-configured QUANT\_RATE variable and the maximum frequency in the document. Any term which has a frequency less than the QUANT value is removed from the list as well. The remaining set is then sent through an MD5 hash to generate a signature, and all documents seen as duplicates under this method are just considered similar to each other. The two user-configured variables, QUANT\_RATE and MIN\_TOKEN\_LEN, are able to be changed from their defaults using the XML-based config file solrconfig.xml.

To take best advantage of this similarity, such as forming virtual hyperlinks, the duplicates detected by this signature need to be kept in. Otherwise, the engine will try to remove duplicates by eliminating all documents that are just similar to each other.

The current configuration simply uses the TextProfile signature with basic defaults. This means that terms made up of two or less characters, and the QUANT\_RATE with a float value of 0.01. Under this QUANT\_RATE, as long as the maximum term frequency in the document is greater than one, then all terms with frequency equal to one will be removed. Schema.xml has been altered to make the index contain a field named "sig", where the signatures are stored. The solrconfig.xml file has been altered to add a new processor, which will activate when an update to the index goes through, such as during Nutch's solrindexer program.

## Link Analysis

Link analysis is the technique used to evaluate the relationships between nodes (links) in information retrieval. Link analysis begins when the crawler has finished crawling and placed url links into a repository. The link analysis tool transverses the url repository and uses information from the urls to generate a webgraph based on the direction of the hyperlinks. From this webgraph we calculate scores for each url to determine its importance in terms of authority.

Nutch utilizes the WebGraph program that is run once all segments are fetched and ready to be processed. Here, the segments are the raw data retrieved by the Nutch crawler. The WebGraph creates three different components: an inlink database, an outlink database, and a node database. The inlink database is a listing of url and all of its inlinks. The outlink database is a listing of url and all of its outlinks. The node database is a listing of url with node meta information including the number of inlinks and outlinks, and eventually the score for that node.[1] Nutch uses LinkRank, which is a PageRank-like link analysis program that converges to find a stable scrore for each url. Similar to PageRank, the LinkRank program starts with a common score for all urls. It then creates a global score for each url based on the number of incoming links and the scores for those links and the number of outgoing links from the page. Once LinkRank completes the url score calculations, the output scores for each url are placed in the node database of the webgraph. Linkrank closely follows the original pagerank formula which is:

LR = (1 - dampingFactor) + (dampingFactor \* totalInlinkScore)

Where totalInlinkScore is the calculated from all the inlinks pointing to a page, taking into account that this is iterative and pages all start off with rankOne score which is (1 / numLinksInWebGraph). For the project we generate virtual hyperlinks between any two pages that are decedents of a common node reached through two different paths. These virtual hyperlinks increase the value of the totalInlinkScore by creating a stronger authority by being siblings to the same parent link. LinkRank has four differences from pagerank. The differences are:

1. There is a spam identification/removal system

2. There is duplicate link limit option

3. The dampening factor, which is set to 0.85, is configurable

4. LinkRank runs for a specified number of iterations(configurable)

## Lucene/SOLR Retrieval Module

SOLR supports two modules for retrieval: Boolean and vector space modules.

### *Boolean Retrieval*

For Boolean retrieval, you can search using exact phrase queries like this: “my exact phrase”. You can also search using a query slop (qs) parameter like this: “runtime exception”~100. That will look for the words {runtime, exception} within 100 words of each other.

Additionally you can do term boosts like this: java perl^3. This will raise the weight of the term perl by a power of 3, over the term java.

These are called query boosts, and it’s how SOLR incorporates LinkRank scores into the program.

#### Vector Space Retrieval

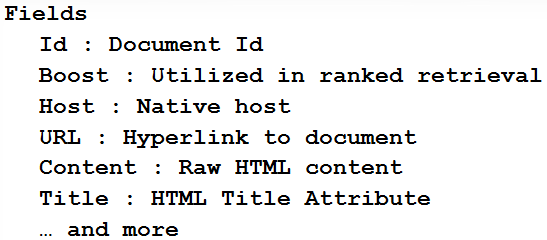
SOLR uses a modified tf-idf weighting scheme. The actual formula is described below. One key difference is it takes into account the LinkRank scores using the t.getBoost() method (where URL’s that documents are sourced from have their PageRank/LinkRank scores taken into account here).



* + **TF**(t in d): term frequency  
    defined as the number of times term t appears in the currently scored document d. Documents that have more occurrences of a given term receive a higher score.  
    = sqrt(frequency)
  + **DF**: document frequency  
    # of docs a term appears in (size of hitlist)
  + **IDF**(t): inverse document frequency  
    rarer terms give higher contribution to the total score  
    = 1 + log(# docs / df + 1)
  + **coord**(q,d): the coordinating factor  
    fraction of query terms that the document contains
  + **queryNorm**(q): normalizing factor  
    makes scores between queries comparable  
    Sum of all of the squared weights in the numerator
  + **Norm**(t,d): Product of several customizable Index time boost factors (field boost, document boost, field length)

## Crawler, Indexer, and Solr Data Schema

One very important configuration that needed to be made was what content needed to be captured during the crawl, indexed, and retrieved. This allowed us to specify which fields we wanted to make use of in our forward document index as well as fields we may want to consider in our retrieval engine. The following is a sample of the fields in our data schema.



## Solr and the User Interface

The web interface is built in HTML, CSS, and JS and makes heavy use of AJAX Solr, a Javascript toolkit that directly interfaces with the Apache Solr API. Any setting that can be configured in the Solr user interface can be directly specified through the AJAX Solr API. This allows us to easily tap into the powerful components of Solr to power our custom interface.

Queries are executed against Solr through client side scripting calls to the server. This is a similar implementation to Google’s AJAX search results. AJAX Solr also allows for pagination of results, which is as easy as passing the page indexes and HTML/CSS styling settings to a pagination widget that comes with the toolkit. These additional requests are also done through client side scripting calls.

Once configured properly, Solr has configurable highlighting functionality. With a single configuration of AJAX Solr, we can access this highlighting functionality and display relevant snippets from the documents we retrieve. This allows us to easily verify that our relevance and retrieval modules are working properly

The design plays off the idea of “catching bugs” which is one of the influences for our seed URL choices for the crawler module. It also makes our interface more aesthetically appealing while remaining simplistic and to the point.

## Implementation Responsibilities

We originally divided up our tasks as follows:

* Crawling: Leith and Revarr.
* Indexing: Josh and Sanner.
* Retrieval/Link Analysis: Joseph and Revanth.
* UI: Josh and Leith.

After we each individually investigated our parts and what tools we could use to do them, or write, we had the following contribution of work to the final product:

* Josh: GUI implementation and initial tools research. Implemented query search conversations and results processing using SOLR. Also determined and configured final indexing schema based on interface requirements which was used for crawling, indexing, and retrieval.
* Leith and Revarr: Devset crawl, then a number of real crawls, seed URL tuning, etc. Also investigated virtual hyperlink injection. Chose Nutch for crawling. Modified regular expressions to fine tune the crawl to weed out spam pages.
* Sanner: Researched and implemented similarity and deduplication methods in SOLR, configuration, tools, procedures, etc. Discovered ways of deduplicating content in Nutch and SOLR.
* Revanth: Studied options for implementing PageRank and Nutch’s LinkRank implementation methods.
* Joseph: Performed initial end-to-end setup of the search engine from crawl to indexing to query lookup with SOLR; crawled a dev site and helped team members get their crawl environment setup. Studied Lucene retrieval and boosting methods; studied Lucene query options such as term boosting.

## Experimental Results

Part of the experimentation came in the crawling phase of the project. We first needed to determine how many iterations of crawling, parsing, and indexing were needed to arrive at the stated requirement of crawling between 1000 and 2000 pages. More importantly, though, we needed to determine which pages to start with so that we provided the user with useful results.

We first started with pages selected from DMOZ under the Computers/Programming/Languages/<language>/ section and either the Documentation/ or “FAQs, Help, and Tutorials”/ sub-sections. It was a bit of a challenge finding sites that had either good information and a good number of out links. After a while, though, approximately 30 sites were selected. We did have to fine tune the seed links as we went. We noticed, in one case, where our Java results were not as useful, so we changed some of the seed links to add more relevant sites.

Once the initial seed list of URLs was compiled, several test crawls were performed. We found that doing a three step iteration of crawl, parse, and index would generally net about 1500 pages crawled and stored. The first pass would include the seed URLs, the second pass would capture approximately 600+ pages and the final run would reach the 1500 page limit. That process would take about an hour to complete.

To add a boost to the link analysis, we attempt to create virtual hyperlinks between urls that descended from the same parent url. By design, Nutch takes utilizes InvertLink to generate a record of links that point to a url only. This, In effect, it shows all the parents of each url and then use these links to inject into the next iteration of the crawl. To this end we modify the WebGraph.java file within the src file of nutch and edit the code to reverse the direction of links and then generate the virtual hyperlinks by adding them into the injection pool. Due to time constraints, this feature was not implemented in the working version of our search engine.

## Discussion

After we discovered our search engine task we discussed ways of improving the state of affairs in the search-engine community with our project (e.g. in order to give a uniqueness to our project). At that point in the semester we had not covered question-answering, latent semantic indexing, relevance feedback, or clustering, so our minds were rather open to experimentation.

We knew very quickly that we all had advanced knowledge in the methods of query tuning and results selection, as quasi-subject-matter-experts in the task of finding relevant articles to our programming related search tasks—as experienced programmers this was very natural to us as we had frequent practice in this. With this knowledge, we were curious as to why some of us could obtain faster results than others, and as to why other less experienced users [in programming, anyways] had a tougher time at finding relevant documents.

One task we attempted was to document our search process, our query formation process, our selection method once results were returned, and any little details about the ergonomics of the search we performed that led to better results. This is one example of this documentation we performed:  
  
 ***The Problem/Error to solve:*** *work.c: In function ‘main’:*

*work.c:8: warning: incompatible implicit declaration of built-in function ‘exit’*

***Query Formation:  
 - ASSUMPTION 1:*** *“..When I search, I only paste in a single line (the second line not the first) because google only gives me a single line search box, so I've been trained to do that.. and really I paste in the longest line containing what seems to me to be the rarest terms that’s unique to the problem I’m having..”  
 -* ***ASSUMPTION 2****:*

*“..as a searcher I know that filenames and line numbers are noise in the search because the filename I chose in my program that generated an error will give arbitrary weight to documents that contain this same filename, which is random and non important to my search; so I will strip this out..”*

***The Query****:* *warning: incompatible implicit declaration of built-in function ‘exit’*

In some of our early analysis, we even documented the reason we avoided clicking on certain links on Google (such as the top ranked ones), with justifications such as these (showing an actual result we pulled from a Google search of that query including preview text which influenced this searcher’s decision):

***GOOGLE RANKED #2***

*incompatible implicit declaration of built-in function 'exit'*

***cboard.cprogramming.com****/.../95528-incompatible-implicit-declarati...*

*Nov 8, 2007 – gcc hello.c hello.c: In function 'main': hello.c:6: warning: incompatible implicit declaration of built-in function 'exit'. Please tell me where am I ...*

***MY JUSTIFICATION:***

*“..Never heard of the website, started with a stacktrace, making me feel like I was about to sift through someone else's error report and spend more time doing that than getting straight to the answer.. RESULT: AVOIDED THIS LINK”*

***RANK 7***

*socket programming error - C*

*www.daniweb.com/software-development/c/threads/223615*

*10 posts - 3 authors - Last post: Sep 18, 2009*

*client.c: In function 'error': client.c:9: warning: incompatible implicit declaration of built-in function 'exit' client.c: In function 'main': client.c:19: ...*

***MY JUSTIFICATION:***

*“..The title of the page (title of the post) didn't seem relevant enough to my search (I wasn't doing socket programming, it wasn't bolded (e.g. it didn't have my search terms in it), and it was very low on the page.. RESULT: AVOIDED THIS LINK”*

And then more notes on the document we finally selected:

***GOOGLE RANKED 3***

*warning: incompatible implicit declaration of built-in function 'exit'*

*www.linuxquestions.org › ... › Non-\*NIX Forums › Programming*

*2 posts - 2 authors - Last post: Aug 14, 2005*

*Hi! Juz got on board! am trying to compile a program and i got the following message... can somebody explain what this following line means? ...*

***MY CLICK JUSTIFICATION****:*

*“..I saw someone talking and describing their problem very straight forward and simply; I saw this first instead of an error trace. Also they were explaining their steps as opposed to just pumping out a stack trace. Also this had a lot of "bolding" in the name, so the bolding made me feel more confident about my choice (e.g. that more of my search term/error was being matched exactly)..”*

*(http://www.google.com/support/forum/p/Web%20Search/thread?tid=2ec0dfba1636c453&hl=en)*

And then we noted the results of how long it took to find our answer upon clicking:

***\*\* AFTER SELECTION, I FOUND*** *\*\**

*“..The website that came up was a bulliten board, and it described the solution in two posts (the other's question and someone's answer). The author suggested adding an include file. That answered my problem and fixed my compilation error. The solution's response for the link I clicked was two lines and was very scannable.”*

***\*\* TOTAL time to find my answer:***

*20 seconds or so*

We also read through a SIGIR paper recommended to us that talked about using a GWAP (game with a purpose) to have many annotators go through a customized search interface with the task of finding answers to certain questions on search engines—using an interface that could capture the links clicked, record times between clicks, and discover, using HMM’s, why certain searchers were better than others.

However, it became very clear after treatment of the material that none of these tasks were trivial and in themselves could have been projects. So in the end, for the purposes of our project, we decided to stick with designing a search engine and do a really good job of making one, rather than trying to adopt a new searching paradigm.

## Conclusion

In conclusion, this project has forced us to get our hands dirty with some very powerful and robust solutions that focus on real IR "best practices" (according to our understanding thus far). This experience has given us a lot of insight into the difficulties of setting up a search engine from scratch, and we now have the tools and understanding necessary to take on domain-specific information retrieval. With further fine-tuning of our system configurations, we feel that our search engine implementation has the potential to be a very strong hub for programmers and software engineers.

Our next challenge would be to take the question and answering portion of our initial proposal and start a more advanced retrieval system that aims to give actual solutions to programming questions.