1 Introduction

The NIMADIVA IR system is part of an ongoing research project in the Department of Computer Science at the University of Maryland Baltimore County. Hence the enhancements to the basic IR engine have been designed with a specific end use in mind.

2 Requirements of the IR System

This section briefly explains the motivation behind the development of our IR engine, and broadly mentions the end purposes of the engine, which acted as a pre-requisite for our design and enhancements.

The overall purpose of the project is to utilize the strengths of machines in building probabilistic models of large volumes of data and the ability of human perception in identifying patterns in dynamic visualizations of the models [1]. The current domain of the project is the academic research world. The project lays claim on the fact that there is a wealth of information about human interaction on the world wide web in the form of research collaborations, corporate fundings, research publications, research assistantships.

In order to formulate patterns of these human interactions, it is necessary to gather as much information as is possible from the web about such interactions. Hence, a specialized IR engine, one that scours the web for relevant information, is the starting point of any such exercise.

There are numerous issues that the IR system needs to address in order for it to accomplish its tasks. Some of these are enumerated below:

- The proposed IR system should be able to, given a set of suitable clauses, narrow its searches over the web for relevant material. In short it should be able to focus-crawl.

- The relevant clauses mentioned above need to be identified. Thus, we must be able to accomplish the task of search parameterization. These clauses may be automatically generated or they may be hard-wired into the IR system.
• As in any model, where the machine as well as the human constantly learns about the process of searching, there must be a mechanism of relevance feedback from the human to the IR engine to let it know if it is performing as it is expected.

• The IR system should not only have the capability to classify pages and information as relevant and irrelevant, but also be able to classify relevant information into predefined categorizes.

• The IR system should be able to provide a measure of temporal changes in pages and such.

3 Project Statement

The requirements of the IR engine described in the previous system may be many and varied. It is clear after an examination of all the issues mentioned above that implementing the whole IR system is out of the scope of the project. Some of the topics are themselves worthy of extensive investigation.

The aspects of the project which were selected to be implemented are the focused crawling nature of the IR system. An automated classifier also can be incorporated for the system to categorize the indexed pages on varying degrees of relative relevance to each other.
4 Phase I: The Indexing Module

Phase I consisted of creation of the Inverted Index generator module. We had Phase III in mind during the design of our indexer. Some of the requirements of the system are as follows:

- The various parts should be designed in a modular fashion so that future changes to the system can be done in a “plug-and-play” manner.
- The inverted index should support incremental as well as parallel indexing.
- The inverted index should be able to routinely garbage collect and reclaim any lost disk space.

4.1 Design

There are four modules which co-ordinate in the indexing. They are:

1. The Filterer Module
2. The Indexer Module
3. The Server Module
4. The Stemmer Module

I. The Filterer Module
The main purpose of the filterer module is to strip HTML tags from the incoming files and produce clean text to be added to the inverted index. As the indexing of documents is not restricted to the umbc-crawl-2 collection used in our project, but will grow to include documents fetched from elsewhere, we had to have our own document indexing system.

The filterer therefore retrieves documents listed in a file and maps them to an internal representation. For indexing the umbc-crawl-2, the filterer used the files-list file. It then strips the documents of all HTML tags, takes out any punctuations, numbers and non-alphanumeric characters. Any hybrid letter-numeral “word” is preserved. The filterer also uses a modified van Rijsbergen stoplist to discard any “common” words.

The filterer then passes the internal document ID and the corresponding clean text to the indexer module. The filterer also saves the internal mapping from document ID to the URL on to a stable storage named as document-map. The interfaces between modules is the stdin and stdout.

II. The Indexer Module
The indexer module is responsible for taking filtered documents and adding them to the inverted index. The indexer accepts an input stream of documents from the filterer module on standard input, formatted in the following fashion:
As each document arrives within the indexer, the length is calculated (to be later used for document length normalization during querying) this information is tabulated and stored in a flat file on disk.

Each of the documents (delimited by the document id and the ascii character 002) are parsed in-memory into a list of postings. Each posting consists of a single term along with the number of times the term occurred within the document as well as the offsets of the term within the document.

Once the postings list has been created, it is handed to a selection module to determine if the document should be added to the inverted index. The selection module is implemented as a shared library that is loaded at run-time, so users can create an inverted index using a combination of selection methods. Two selection modules were implemented for this project. The first is named 'offline' and it determines that every document should be indexed. This module is used when indexing umbc-crawl-2 for the purpose of running benchmarks. An alternative selection module is also implemented named 'profiler'. The profiler selection module compares the document to a set of profiles (represented as XML documents) to determine if the document should be added. If the document has a similarity to any one of the profiles above a particular value, it is marked for indexing; otherwise the document is simply discarded. Additional modules can be easy developed and used with the indexer to provide additional flexibility to the indexing process.

The documents that are marked for insertion are added to the inverted index. The inverted index is implemented in an abstract way so that the implementation details of the actual storage structures of the inverted index are hidden from the indexer; any sort of data structure could be used to store the inverted index without having to modify the indexer as a result. As speed and fault-tolerance were important considerations, the Berkeley DB (version 4.1.24) was used to store the inverted index on disk. The Berkeley DB has the advantage that a separate lexicon is not necessary, and as a result, the system may be stopped and restarted arbitrarily without requiring existing data to be re-indexed. In addition, the Berkeley DB caches data in memory, allowing future reads fast - thus providing a good compromise between speed and reliability.

For each term that has been added to the inverted index, there is stored a list of postings. The postings are stored by converting the document identifiers into document gaps and then encoding the postings using variable-byte encoding.

III. The Server Module
The server was designed so as to enable multiple filterer modules working on separate machines to send their clean text to the server. The server would then assign document ID’s
to each of the incoming documents and stream it to the indexer module.

The server originally handled the assignment of document ID's by creating an MD5-hash of the document contents. The server would check if the document had been seen before by searching through the MD5-hash entries of existing documents for duplications. This way different URL's pointing to the same document would not be indexed twice. Furthermore, if pages changed over time, their MD5-hash entries would also change, thereby ensuring a separate document ID for the changed page. This was implemented in order to track and index documents changing over time.

The server was originally coded in Java, which was then compiled into binary code by gcj, the GNU Java-to-native code compiler. However, during our test run, we realized that the server was not a lightweight enough process we had hoped it would be. It started to become the bottleneck in our indexing process. Therefore, it was temporarily disabled for this project.

IV. The Stemmer Module

The stemmer was originally slated to be used by the server module before the words were passed to the indexer. It is to be noted at this point that we implemented Porter’s stemmer for this module. We however, did not port it over our non-server based system.

4.2 Benchmarks

Following are some of the results we received from our indexing process. The phase1-benchmark program automated most of our Phase I benchmarking.

1. There are a total of 241,309 words in our collection.
2. The /data/nicholas2/ian/umbc-crawl-2/files-list file lists a total of 82,385 documents.
3. Our index contains words for 82,317 documents.
4. There are a total of 7,928,125 postings in our index. This does not include the offsets which are also stored.
5. The longest index contains 44,384 postings, the term being umbc.
6. The shortest index contains 1 posting, the term being rehbehn.
7. There are an average of 32.85 postings per term stored.
8. It took us about 10:54:00 hours to generate the whole index.
9. Our memory usage reached a peak of 370 MBytes during the indexing process.
10. Our index.db on disk inverted index, which also contains the last Document ID for each word occupies a total of 82,718 MBytes.
11. The document-lengths file occupies about 772.7 KBytes.
5 Phase II: The Query Module

This is the query module which searches the inverted index for queries.

5.1 Design

We implemented the Vector Space Model for our querying. The query terms are taken in one at a time; their inverted indices (if they exist) are retrieved from index.db.

The inverted index for each term is then uncompressed and the document gaps are converted to document ID’s. The terms are then sorted with the term appearing in the least number of documents going in first. These terms are then successively processed.

For each inverted index, the term frequency and the inverted document frequency terms are then calculated for each document it is contained in, and the documents are then ranked. The final tf-idf product-sums are divided by a measure of the document lengths to form the final ranks.

There is an upper limit to the number of documents which are processed. All documents beyond a certain number are discarded. However, we finish ranking all the documents within the upper limit for all the terms. The query module then normalizes the ranks and returns a list of URL’s.

NOTE: We discovered in course of experimentation with our querying module that the lengths of documents were having an adverse effect on the ranking. Long documents were “unfairly” being overly penalized.

We therefore suppressed the dividing factor by using \((1 + \log (\text{doc.length}))\) instead of \(\text{doc.length}\).

5.2 Benchmarks

Following is the benchmarking results for Phase II.

A.

1. We used “procmail” for our querying.

2. The time required for our query model to rank 100 documents and display 20 documents was 5.03 seconds.

B.

1. We used the topic 20: procmail for our topic query.

2. The query based on the title section was procmail.

3. It took us 9.39 seconds to rank the top 100 results.

4. The query based on the description section was this topic is in reference to the unix program procmail.
5. It took us 9.74 seconds to rank the top 100 results.

6. The query based on the narrative section was relevant information is information about procmail tutorials and recipes for use with procmail itself.

7. It took us 10.93 seconds to rank the top 100 results.

C.

![Recall Precision vs. Number Of Documents Ranked](image)

Figure 1: Recall Precision vs. ranked documents for returned results

1. We used automated query for this part. The query-retriever module parses through the umbc-topics.txt file, and retrieves the topics corresponding to the <title> section. These are then converted into clean text, and passes them on to the query-returner module. This is just a wrapper around the basic query backend which feeds the queries, retrieves the results, formats them to the trec_eval form and then writes them out.

2. The above modules were run, and the results were redirected to phase2.results.nimadiva, which was submitted.

3. We ran trec_eval on our results. Returning the top 100 results after ranking the top 100 documents got us a recall precision of 0.1746. The average recall for the same run was 0.1557.

We ran a set of experimental results for recall precisions. These are summarized in the Table 1 and Figure 1. Table 1 provides the recall precisions for the returned results in columns versus the ranked documents in rows. Figure 1 gives recall precisions versus
<table>
<thead>
<tr>
<th>#Ranked</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Disp’ed</td>
<td>20</td>
<td>0.1732</td>
<td>0.2163</td>
<td>0.2180</td>
<td>0.2110</td>
<td>0.2111</td>
<td>0.2164</td>
<td>0.2149</td>
<td>0.2157</td>
<td>0.2182</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.1746</td>
<td>0.2163</td>
<td>0.2184</td>
<td>0.2133</td>
<td>0.2139</td>
<td>0.2141</td>
<td>0.2154</td>
<td>0.2171</td>
<td>0.2175</td>
</tr>
</tbody>
</table>

Table 1: Recall Precision vs. ranked documents for returned results

number-of-documents-ranked. The recall precisions for when the top 20 and the top 100 documents are plotted.

As expected, the recall precision for 100 documents were slightly higher than for 20 documents. However, because of the dearth of relevant results from umbc-qrels, the difference is not as pronounced as one would expect it to be. However, we did observe a curious phenomenon.

The recall precision initially rises sharply as we increase the number of documents ranked. However, it reaches a peak at around 300 ranked documents. The recall precision then drops off until about 450 ranked documents, after which it again steadily starts rising again. We were unable to explain whether it was a feature of our collection or as a result of the umbc-qrels. We thought it important enough to report our results.

We ran into some problems when trying to extract the results from the trec_eval. We deemed it necessary to report this. There are inconsistencies between doc-files-only and the corresponding entry in files-list. This led to some confusion, before the source of the problem was diagnosed. We converted the doc-files-only into a consistent format as a local copy.
6 Phase III: The Focused Crawler with Automated Profiling

The NIMADIVIA IR engine has been designed to generate automated profiles which represent search-threads. Based on these profiles, the Focused Crawler module searches the web for documents deemed relevant to the search-threads.

We discuss the Profiler and the Focused Crawler modules in the next two sections.

6.1 The Profiler

The NIMADIVIA profiler automatically generates profiles based on queries. These profiles exist as XML documents.

Since the profiler is directly tied to the queries, the query module was modified to update the profiles automatically. The XML profiler is executed as a shared library which is linked during compile time. The profiles store the query words. In addition, it also stores the top \( n \) document URL’s retrieved during the current search. An example profile generated from one of the topics in \textit{umbc-topics.txt} is provided below.

```xml
<?xml version='1.0'?>
<profile name='004'>
  <relevant-documents>
    <document id='40618' relevance='1.000000'/>
    <document id='40622' relevance='0.959262'/>
    <document id='40617' relevance='0.901620'/>
    <document id='40620' relevance='0.883521'/>
    <document id='40621' relevance='0.883521'/>
    <document id='40619' relevance='0.883521'/>
    <document id='32880' relevance='0.797376'/>
    <document id='43768' relevance='0.729124'/>
    <document id='56483' relevance='0.726080'/>
    <document id='32877' relevance='0.692189'/>
    <document id='43216' relevance='0.671332'/>
    <document id='33948' relevance='0.665128'/>
    <document id='32879' relevance='0.613178'/>
  </relevant-documents>
  <relevant-terms>
    <term>alcoholism</term>
  </relevant-terms>
</profile>
```

The proposed merging algorithm of the profiles after each query, searches all existent profiles for similarity. The measure of similarity is given by the commonality of the top \( n \) documents retrieved by the query, and stored by the profile.
6.2 TheFocusedCrawler

The NIMADIVA focused crawler is intended to use the live web in order to retrieve documents it deems are relevant for retrieval. The crawler carries a set of profiles containing key information on what documents are relevant in order to make such decisions.

There are two basic components to the focused crawler - one that retrieves documents, and one that deems the documents relevant or irrelevant to its current set of profiles.

To retrieve documents, currently `wget` is used. `wget` will download a set of documents that can then be checked for relevance, and a set of these documents will be selected for indexing. Work is planned to make a complete and intelligent crawler to replace the use of `wget`. The crawler would be able to use its profile information to make a determination on where to find the most relevant documents on the web, and crawl those specific locations, eliminating the processing of excessive amounts of irrelevant data.

The second portion of the focused crawler attempts to make a decision as to whether or not a given document is relevant to a known user information need. After receiving a new document from the web, the crawler processes the document, creating an inverted index from it.

The crawler continues by comparing the new document against all of its profiles that it has been assigned. The document is compared to the profile using a simple VSM model in which the profile terms are treated as query words. The cosine similarity is derived between the new document and the profiles. If this similarity falls above some defined threshold, the crawler makes a recommendation to index the document, and add it to the index repository. If the similarity of the new document does not match the given threshold level with any of the profiles assigned to the crawler, then the document is given the recommendation to be discarded.

Because the focused crawler currently works on disk, rather than retrieving documents straight from the web, it is important to recognize the above use of the term "recommendation". Currently, the focused crawler would inform the user that a document is or is not relevant, and the user can choose to take further action based on this recommendation. Once live web crawling begins, the user will likely be removed from this process, allowing for the crawler to cache documents it believes to be relevant, and discarding those which are not.

We experimented with some profiles generated with queries on topics in `umbc-topics.txt`, and crawled the web for documents relevant to the profiles. An example list of relevant files retrieved is given in Appendix A. The corresponding profile was provided in the previous section.

7 Conclusions and Future Work

We succeeded in implementing aspects we sought to achieve. Phases I and II were successfully implemented and benchmarked within the scope of this project. Preliminary experiments for Phase III were also performed. There still remain some issues regarding our implementation, which need to be addressed in the future.

The server’s design although good proved to be too heavy weight. One of the solutions is to write the server in C. The server also needs to be multi-threaded. In our experiments, we
found that individual indexers spent too much time waiting in queue to submit their results to the server, causing a bottleneck in the process.

The query front end also needs to be finished. We assigned it low priority because of more ponderous design implementations facing us during the project. Besides, the query back end was made simple and modular enough so that various front ends could be experimented with. Although we did not incorporate phrase-matching, we stored offsets in the inverted index so that the query module can be upgraded without any changes to the indexing module.

Another one of our alternate goals was to implement the filtering part in C rather than in awk filterer being used currently. This would entail simple function calls from the indexers rather than waiting for `B-separated documents inputted through stdin. This makes the Phase I module less error prone.

The crawler presently works in an unintelligent fashion, caching all documents it encounters and making the decision off-line. A better and faster crawler which makes decisions in a live fashion while it follows the links would make the retrieving module faster. Another thing which can be done is to experiment with the different levels of threshold at different times of crawling, and examine the effect of profile growth with the threshold values required to fetch relevant documents.

There is also the issue of profile merging. Currently each query creates its own profile. A design decision remains to be made at the similarity measurement scheme for different profiles. A more important decision is the issue of merging the document ID’s of two profiles together while keeping the number of documents in a profile under a certain threshold.

While there remain hurdles to be overcome, we feel that we have essentially laid the groundwork for future research work on the NIMADIVA IR engine.
A List of Relevant Crawled Files

The List of crawled files for the “Baltimore Ravens” profile (listed in Section 6.1) that were deemed relevant.

http://www.nfl.com/news/00aflpomdec.html
http://www.nfl.com/tvradio/nflfilms/hardknocks072401.html
http://www.nfl.com/radio/chron00.html
http://www.nfl.com/ce/multi/0,3783,5576155,00.html
http://www.nfl.com/ce/multi/0,3783,5631281,00.html
http://www.nfl.com/ce/multi/0,3783,5626038,00.html
http://www.nfl.com/ce/multi/0,3783,5607800,00.html
http://www.nfl.com/ce/multi/0,3783,5590676,00.html
http://www.nfl.com/ce/multi/0,3783,5584447,00.html
http://www.nfl.com/ce/multi/0,3783,5582065,00.html
http://www.nfl.com/ce/multi/0,3783,5569623,00.html
http://www.nfl.com/ce/multi/0,3783,5552888,00.html
http://www.nfl.com/ce/multi/0,3783,5550105,00.html
http://www.nfl.com/ce/multi/0,3783,5544731,00.html
http://www.nfl.com/ce/multi/0,3783,5544590,00.html
http://www.nfl.com/ce/multi/0,3783,5452929,00.html
http://www.nfl.com/ce/multi/0,3783,5426364,00.html
http://www.nfl.com/ce/multi/0,3783,5420985,00.html
http://www.nfl.com/ce/multi/0,3783,5388152,00.html
http://www.nfl.com/ce/recap/0,3762,NFL_20020809_DET@BAL,00.html
http://www.nfl.com/ce/recap/0,3762,NFL_20020815_NYJ@BAL,00.html
http://www.nfl.com/ce/recap/0,3762,NFL_20020823_BAL@PHI,00.html
http://www.nfl.com/ce/recap/0,3762,NFL_20020829_BAL@NYG,00.html
http://www.nfl.com/teams/BAL.html