1 Webcrawler

- URL frontier - list of seeds
- DNS lookup
- Content seen - shingles
- Politeness - Robots.txt and delay
- Duplicate URL eliminator

Our web crawler is based around a Crawler class, which has an instance of the url frontier.

The Crawler class instantiates the url frontier with a seed consisting of around 8 urls.

Furthermore the crawler contains 2 dictionaries and a Parser object.

The two dictionaries

1. Keeps track of when a host can be visited, using the ip address of the server at which the given webpage is hosted, and the time at which the server can be visited again.
2. A map used to cache the robots.txt file of the different hostnames.

When the crawler is started it fetches the queue from the url frontier and for each url it starts by requesting the robots.txt file for the hostname. If the url is valid according to the robots.txt file we continue by ensuring that enough time has elapsed since last visit, this is achieved by an DNS lookup. When both these conditions has been fulfilled we request the web page and extract all anchors on the page and add them to the queue of the frontier. Lastly we store the web page to be retrieved later by the indexer.

1.1 Url frontier

The very basic understanding of the responsibility of an url frontier is to keep track of a queue of urls is going to be crawled. The frontier should implement prioritization and politeness. Prioritization in form of importance of a given webpage and politeness in form of respecting the robots.txt file for each host, and be polite in the way of sending a request to a host may only happen once every X seconds.

Our implementation

Our solution is very simple. The frontier class has an queue and a dictionary.

The queue contains all urls which is going to be crawled. and the dictionary contains all pages which has been crawled. To ensure that we do not add an url to the queue which has already been crawled.

Our frontier do not implement politeness this responsibility is in our implementation given to a class called Crawler.

We have implemented simple prioritization in form of first in first out queue, meaning that every page is equally important.

Proper implementation

A proper implementation of the frontier would have the responsibility of sustain politeness and a prioritization of the webpages.

This can be achieved by using front queues and back queues.

The front queue implements prioritization by assigning an integer priority to an url and adding the url to the front queue of the given priority.

The heuristics of assigning a priority can be based on how often a page is refreshed (this is based on information from previous crawls) and/or the prioritization can be application-specific(e.g. news sites should be visited more often).

The back queue implements politeness by keeping a heap updated by adding an element to the heap for each queue consisting of an the queue's id and a time at which an element from the queue can be visited again.

By using a back queue selector it is possible to achieve this. A crawler requests and url from the back queue selector, which extracts the root of the heap (which is the queue containing the host to be crawled next). The back queue selector then fetches the head of the queue, and checks if the queue then is empty. If it's empty it pulls an url from the front queues, and if there is already a back queue for the url it is appended to this back queue, else it is added to the empty queue and an heap entry is added.

The invariants for a back queue is that only a single host is allowed for each queue, and can never be left empty.

1.2 Content seen - shingles

We are fetching each website after confirming with the corresponding robots.txt file. When we have fetched the page we extract the anchors of the page and store the page.

Improvements

It would be a great improvement to check if the content has already been seen. This is achieved by using shingles. Shingles is a way of splitting the content of a page into parts of words. Then by comparing the shingles of two documents we can by using Jaccard similarity decide how similar the two documents are:

$$Jaccard(A, B) = \frac{A \cap B}{A \cup B}$$

Depending on the result we can tell if they are similar. E.g. a similarity of 0.9 can tell that the content is equal.

An optimal duplicate-content implementation, hashes the shingles with 84 different algorithms. For each algorithm we store the minimum hash values. We then look through all pairs of minimum hashes and our Jaccard(A,B) is then the amount of times the hashes are equal.

1.3 DNS Lookup

We use DNS lookup to ensure politeness and not spam a page with requests. At the moment our solution is synchronously. This means that when we request a dns lookup we wait till we get a response. This could be heavily improved by doing this asynchronously since if we do not get an answer from the dns server we will wait for a time out. By doing this asynchronously we could have a thread looking up DNS' and add them to a map as they get added to the queue of urls. Then when we pull a url from a queue we can immediately check if we are allowed to visit the server by looking up the host in the map.

1.4 Normalizing urls

When we extract urls from a webpage we are normalizing them before adding them to the queue.

When normalizing we want to ensure that

- Urls are formatted properly (https:// or http://) if the protocol is not indicated we append (http://).
- Relative urls are resolved
- Remove query part of an url (Everything of the url after an '?' including the '?')
- Removing bookmarks from the url (Everything of the url after an '#' including the '#')
- If the url starts with # we are simply returning the domain of the page.

Improvements

To improve the normalization process we should filter the urls and exclude the urls which we do not want. This could be images, links starting with ftp://. Also we should remove default pages such as "index.html" or "default.aspx". At the moment www.somepage.com and www.somepage.com/index.html appears as different pages in our queue and will be handled in our application as different pages.

2 Indexing

- Information retrieval (Boolean retrieval and Inverted index)
- Normalization
- Tokenization
- Stopwords
- Stemming

2.1 Information Retrieval

Retrieval of a website means finding documents that satisfies the needed information. To provide documents that matches the requested information, we need to analyze the content of the document. This can be done using boolean retrieval or an inverted index.

2.1.1 Boolean Retrieval

For each document we save the words that occur. The words are represented in a table containing all words occurring in all documents. Fx. two documents is represented like this:

Word	Document_1	Document_2
hello	1	0
welcome	0	1
world	1	1

This means that "hello" and "world" occurred in the document, but "welcome" did not. Using this method, the number of occurrences of words is not saved. This leads to another method that represents the data better.

The following table shows a better representation of the words in the document:

Word	Document_1	Document_2
hello	3	0
welcome	0	1
world	3	1

Even though this representation is better, the matrix used to represent it is very sparse leading to increased storage use. An improved representation is to use an inverted index.

2.1.2 Inverted Index

In an Inverted Index we save the number of occurrences of each word in each document. We do not represent if a word does not occur in a document since this is a waste of storage. Fx. if we have document_1 "Hello world, Hello world, Hello world" and document_2 "Welcome world" it is represented as follows:

Word Document ID : Number of occurrences

hello 1 : 3 welcome 2 : 1 wolrd 1 : 3 2 : 1

This way we save the storage of all the boxes in the matrix where the value is 0. Also it is only necessary to have one Inverted Index, where it is necessary to have boolean retrieval table for each document.

wolrd 1 : 3 2 : 1

This is called a posting list for the word "world", and one element in a posting list is called a posting.

3 Normalization

Normalization is about making words that can be spelled differently look alike,

Fx. USA and U.S.A. both means the same, hence it should be interpreted as the same word. This also goes for sugar free and sugar-free etc. A sentence containing "and/or" will have the forward slash replaced by a whitespace such that it results in "and or". If there is more than one whitespace it is replaced by a single whitespace.

All text from documents and queries are normalized/sanitized using the following methods which is a sequence of regular expressions. The first converts all letters to lowercase and removes html tags, which has not been removed. This means that it is not possible to do a query for i.e. "¡html¿".

The second replaces "/" with a white space. The third replaces multiple spaces with a single space.

The fourth and last removes all characters which are not a letter, a number or a white space. Finally we remove any starting and trailing white spaces.

```
1 public string SanitizeText(string input)
2 {
3   input = Regex.Replace(input.ToLower(), @"<[^>]*>", " ");
4   input = Regex.Replace(input, @"/", " ");
5   input = Regex.Replace(input, @"\s+", " ");
6   return Regex.Replace(input, @"[^a-z    0-9\s]", "").Trim
        ();
7 }
```

3.1 Tokenization

This part of the indexer takes a string of text as input and outputs all words one by one. Every word is called a token. We split the input string on whitespaces making our tokenizer returns a list of tokens.

3.1.1 Stopwords

The indexer can be configured to exclude certain words from the documents, fx. "the", "it", "is" etc.. These words are excluded because almost every document contains the words, and therefore it is irrelevant to the result of the search.

Stopwords are excluded from both our documents and search queries. This means that a query for the movie "It" will not yield any result in our search engine. And the query "It the movie" will only show results for the term "movie" since the other two are marked as stop words.

We have implemented the default english stopwords list from http://www.ranks.nl/stopwords which contains 176 stopwords.

3.1.2 Stemming

A word can occur in different conjugations, fx. "connect", "connection", "connector", "connected" etc.. Therefore the stemmer removes some of the word,

such that different occurrences of "connect" is represented as the same term after the stemming is complete.

We have implemented The Porter Stemming Algorithm which is created by Martin Porter, and it is most commonly used stemming algorithm.

4 Content based ranking

When computing results for a query, it is necessary to order them according to some measure to minimize the result set. Ranking is the process of attaching a score to each document (in order of importance) and then return top n number of matches, i.e. the documents that best matches the query.

Ranking can be done according to various criteria, fx content based (vector space model or probabilistic model), where the result set depends on the specified query. Other criteria also exists such as applying heuristics to the query or using structure-based ordering techniques (PageRank and HITS). Now, we will consider the implementation for the mini-project, which is using a content based ranker.

Our ranker is based on Term Frequency and Inverted Document Frequency, which are used to rank a document according to how many times a term is found in that document and how many times in all documents a term is found. The approach helps giving high weight to terms that are likely to the most important according to the query.

4.1 Term Frequency (tf)

The tf is defined as the number of times a term is found in a document. The reason for considering tf is to give weight to frequently found terms in a document, since frequently occurring terms often signifies the relevance of that term. One drawback though is that keyword-stuffing can negatively impact the output.

4.1.1 Normalization

We have decided to use a log-frequency weighting of the term frequency. This allows us to dampen the effect of terms that are found most frequently (ER DET HELT KORREKT?).

To be able to avoid taking the logarithm of a tf of 0, we are normalizing the term frequency in the GetNormalizedTf() method. If the tf is 0, we just return 0 as there are not weighting to do, otherwise we return 1 + Math.Log10(termFrequency); where we simply add 1 to ensure the expression not evaluates to 0, since this would also yield 0 when calculating the Tf-Idf..

It should be noted that this normalization is not true normalization. A better approach to normalization is to prevent a bias towards longer documents. To overcome this, we should have divided the tf with the sum of the number of occurrences of all terms in the document.

4.2 Document Frequency (df)

Document Frequency is the number of documents that contain the term in question. Most often, not all terms in a query is of equal importance, since rare terms can add more information to the query. Consider fx a query using lots of stop words. The df is thus a measure for how rare a query term is, and since stop words are very common in most documents, it is desirable to give more attention to terms that are found less frequent in these.

4.3 Inverse Document Frequency (idf)

Defining the inverse of the df is simply a measure of giving rare occurrences of a term in a document a higher weight. If N denotes the number of documents in a collection of documents and the inverse document frequency is defined in terms of I, we can define the idf as log(N/I). As with log-frequency weighting, the logarithm is also used here to give less weight to terms occurring frequently in a document and high weight to terms occurring rarely in documents.

As with tf, the idf is also normalized in the GetIdf() method, where we return Math.Log10(docCount / (double) docFrequency);. This ensures that the effect of the idf is also dampened when the docCount is high and the docFrequency is low.

4.4 Term Frequency - Inverse Document Frequency (tf-idf) weighting

Is simply the product of the tf and idf weights.

4.5 Ranking algorithm

This algorithm does the actual ranking of terms in documents. It returns a list of key-value pairs with the key being the document id (int) and the value being the weight (double).

The algorithm begins by iterating over all the supplied query terms for a query. If the query terms not are in the list of all terms, we return null to avoid getting NullPointerExceptions. It proceeds by looking through all postings for a query term to calculate its tf-idf weight. This weight is added to a sortedlist (sorted by value, the weight), which is then returned when it have iterated through all query terms.

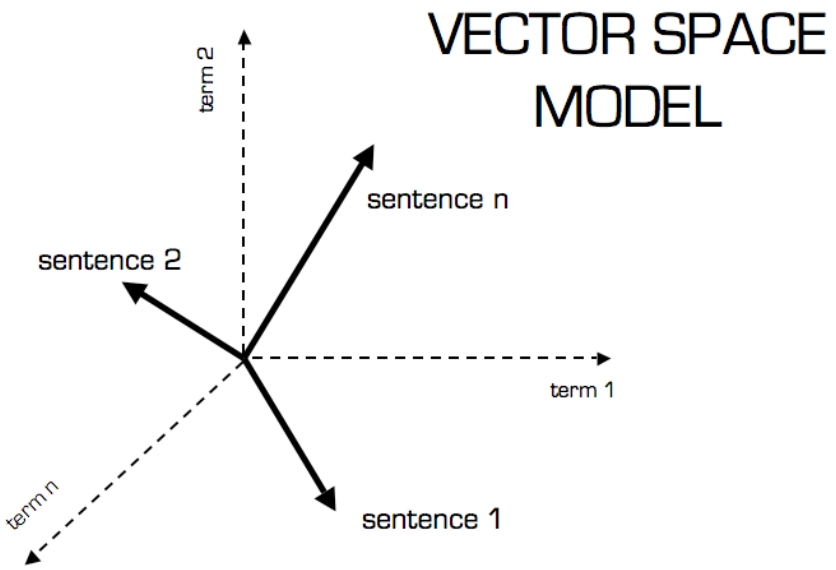
```
1 public IEnumerable<KeyValuePair<int, double>> Rank(
      IEnumerable<string> queryTerms, SortedDictionary<string,
      Term> terms, int documentCount)
2         {
3             SortedList<int, double> scores = new SortedList<
                  int, double>();
4             foreach (var queryTerm in queryTerms)
```

```
5               {
6                   if (!terms.ContainsKey(queryTerm))
7                       return null;
9                   Term currentTerm = terms[queryTerm];
10                  int documentFrequency = currentTerm.
                        Frequency;
11                  foreach (var posting in currentTerm.Postings
                        )
12                  {
13                      int docId = posting.Key;
14                      int termFrequency = posting.Value;
15                      double weight = GetTfIdf(termFrequency,
                            documentCount, documentFrequency);
16                      if (scores.ContainsKey(docId))
17                      {
18                          scores[docId] += weight;
19                      }
20                      else
21                      {
22                          scores.Add(docId, weight);
23                      }
24                  }
25              }
27              return scores.OrderByDescending(kvp => kvp.Value
                    );
28          }
```

Optimized Implementation

Other variants for ranking in information retrieval systems also exist. Our implementation of tf-idf weighting does fx not normalize the vector space model. The most widespread method is to use the cosine similarity meassure, where two vectors (documents in the vector space) are "dotted" together. The result of the dot product is a scalar value that if divided by the norm of the vectors, gives the angle between these. This makes it ideal to be used as a similarity score for our documents. Similar documents will then have a score near 0 (a small angle), "orthogonal documents" (or vectors) will then have a score of 0. Vectors pointing in opposite directions are -1.

$$cos\theta = \frac{d_2 \cdot q}{\| d_2 \| \| q \|}$$



Other methods to consider

- Pruning - the process of filtering non-contenders
- stop words
- Eliminate query terms from postings with low idf scores, as they have a high number of documents and do not add much to the result set.
- champion lists
- Precompute the "top documents" in the query terms postings. Then on query time only compute scores for documents in the champion list and pick the top of these.
- Clustering
- - is the process of grouping documents into clusters according to how they relates to each other.
- Prepossing when using cluster pruning
- Documents are divided into leaders and followers
- Pick sqrt(N) leaders at random and for any other document (a follower), compute its nearest leader.

- Query processing when using cluster pruning
- For a query, find the nearest leader L (document)
- Select K nearest documents from L's followers
- Zone Indexes
- Assign importance according to the semantic meaning of the contents of a document. Fx, headers and paragraphs in a html document might be more meaningfull than the text extracted from an unordered list.

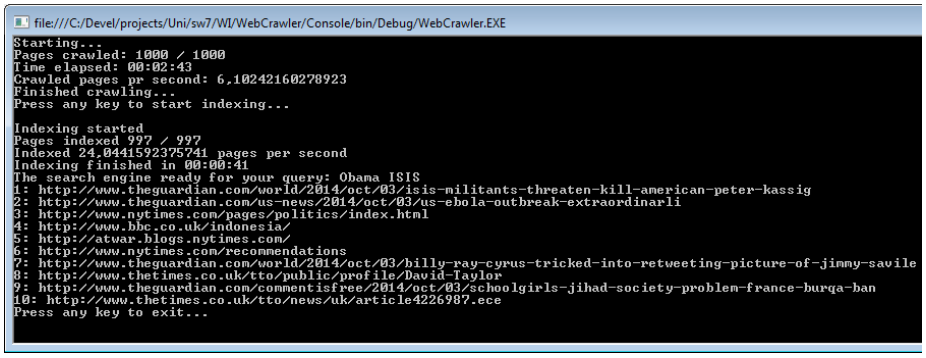
4.6 Query demonstration

In the following image a query is demonstrated.

We are not able to query while crawling, furthermore our search engine is very linear. We are first crawling a number of pages, then we are indexing the pages and then we are ready for a query.

This structure of our search engine means that we are not able crawl an unlimited amount of pages. Our search engine is only functioning with a set limit.

The query "Obama ISIS" gives us the following top 10 results:



As you see in the query results we are getting some kinda weird results compared to the query. This is because of the structure of the web pages crawled and the way our indexer analyses the pages. We are simply removing all html tags, this means that all the text appears to be one large text, even if it is not.

As an example take query 7. This page is shown in the picture below. Here the actual story of the article is in the left side of the webpage, but on the right side, there are links to other articles. An as the red circles show "obama" and "isis" is actually not even close to each other.

This page should definitely not appear in the search result. The reason this page gets ranked this high is because of the amount of pages we have crawled. It only contains each query term once it is in the top 10 pages, but because of the small amount of pages indexed this gets ranked high. Probably there are many query results like this example which are not in the top 10.

Edition: UK | US | AU | Sign in | Beta | About us | Today's paper | Subscribe

This site uses cookies. By continuing to browse the site you are agreeing to our use of cookies. Find out more here

theguardian

Google™ Custom Search | Search

News | Sport | Comment | Culture | Business | Money | Life & style | Travel | Environment | Tech | TV | Video | Dating | Offers | Jobs

News › Technology › Twitter

Billy Ray Cyrus tricked into retweeting picture of Jimmy Savile

Is Billy Cyrus the new Donald Trump? The country singer has fallen for the same prank as the business mogul

Elena Cresci

Follow @elenacresci | Follow @guardiannews

theguardian.com, Friday 3 October 2014 14.28 BST

Billy Ray Cyrus has been caught out on Twitter, accidentally retweeting a picture of Jimmy Savile. Photograph: RICHARD DREW/ASSOCIATED PRESS

First Donald Trump, now Billy Ray Cyrus – attempting to get celebrities to retweet pictures of notorious criminals seems to be the new thing.

The American country star managed to fall for exactly the same trick as Trump on Friday by retweeting a request from a follower. @ekajronnoco claimed his granddad "is a huge fan" and would really love a retweet for his birthday.

Billy Ray Cyrus retweeted

Jake Imo @ekajronnoco · 13 hrs

@billyraycyrus Hi pal, it's my Granddads birthday today and he's a huge fan. He'd love a RT!

Share 1264 | Tweet 303 | +1 8 | Share 1 | Email

Technology
Twitter

World news
Donald Trump

Music

Media
Social networking · Jimmy Savile

UK news

More news

Guardian Shorts: *Connected*

From Barack Obama a dusty box of books in Cardiff, *Connected* is about the transformative power found at the junction of online and offline communities.

Click here to read a free extract and buy the

Today's best video

Hong Kong pro-democracy protesters clash with pro-government supporters
Hong Kong's pro-democracy umbrella movement comes under attack from pro-government counter-protesters
38 comments

Father of Briton held by Isis makes appeal
Father of UK journalist John Cantlie asks the Islamic State to release his son

Top Gear driven out of Argentina
Cast and crew of Top Gear pelted with stones in town of Tolhuin

The album to hear this week
Alexis Petridis recommends The Breaks by Martin Carr, formerly of the Boo Radleys
19 comments

Seven digital deadly sins