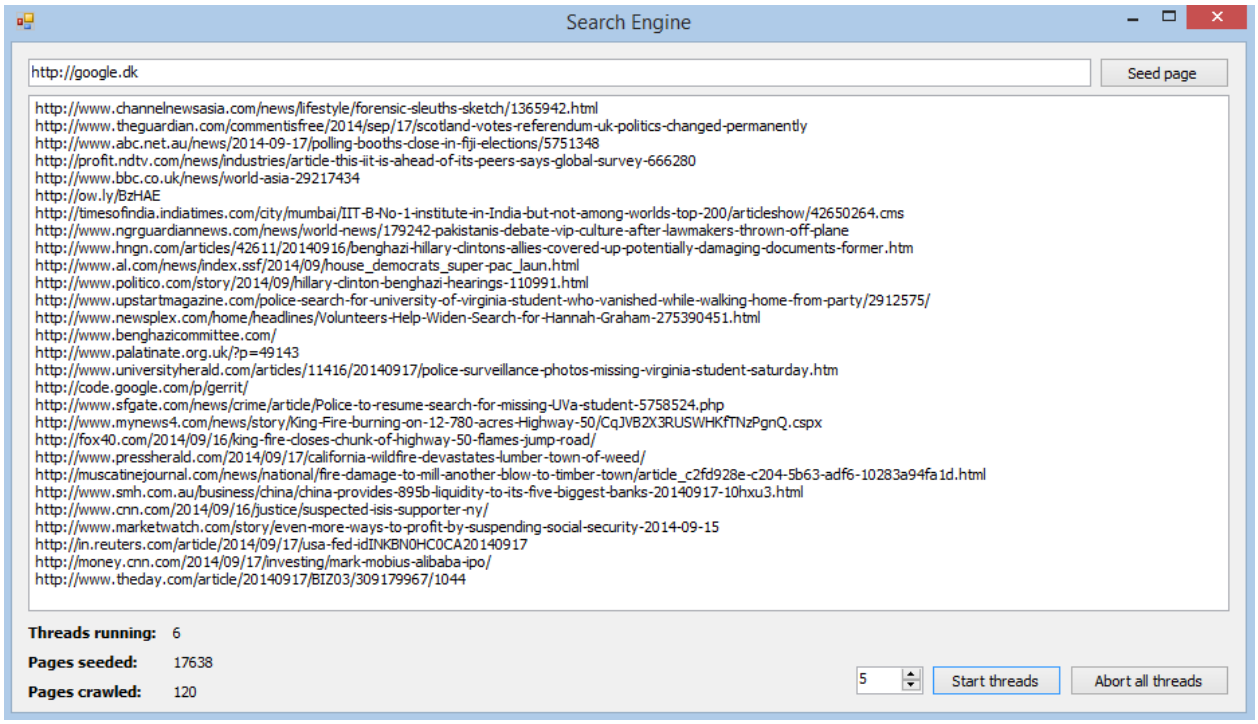
# Web Intelligence

Mini Project 1 - Crawler

I have implemented a multi-threaded web crawler as a Windows Forms application. The application features a “page seeder”, statistics about the crawling and input controls for spawning new threads.




## URL Frontier

To ensure a host is not hit too frequently I store a dictionary of hosts and information about the last hit. Whenever a host is hit the entry is updated with a new “HitInfo” and the responded boolean is checked when the host responds. Using a dictionary with the host as the key ensures efficient look-up. Implementation of “hosts hit”:

```
private Dictionary<string, HitInfo> hostsHit = new Dictionary<string, HitInfo>();

public class HitInfo
{
    public HitInfo()
    {
        Time = DateTime.Now;
        Responded = false;
    }

    public DateTime Time { get; set; }
    public bool Responded { get; set; }
}
```

When a thread finishes crawling a web page the thread asks for a new page to crawl using the method “GetNextPage()”. If the host of the page has been hit before the “HitInfo” is examined to ensure the server has responded and the minimum delay between hitting a host is met. If this is not the case the page is re-queued. Shared resources are locked in order to ensure mutual exclusion.

The implementation ensures politeness because no host is hit twice within 5 seconds from a response. If the pages seeded only include links to the same host the threads will re-queue the same pages in circles until the minimum delay is met. However, this case is very unlikely!

```
private int minimumDelay = 5;

private Uri GetNextPage()
{
    Uri uri;

    lock (queue)
    {
        if (queue.Count == 0)
            return null;

        uri = queue.Dequeue();
    }

    lock (hostsHit)
    {
        string host = uri.Host;

        if (!hostsHit.ContainsKey(host))
        {
            // If we have not hit the server we wil just go ahead and hit it
            hostsHit.Add(host, new HitInfo());
            return uri;
        }
        else if (hostsHit[host].Responded && hostsHit[host].Time > DateTime.Now.AddSeconds(minimumDelay))
        {
            // If we have hit the server before but the server has responded and the minimum time of delay
between hits is overrun
            hostsHit[host] = new HitInfo();
            return uri;
        }
        else
        {
            // If we are not allowed to hit the page right now it is reseeded in the queue
            SeedPage(uri);

            // The method is called recursively to get another page
            return GetNextPage();
        }
    }
}
```

## Retrieving links

I use a regular expression to retrieve the ”href” HTML attribute from the page being crawled. All links are converted to lower-case and relative links are converted to absolute URLs.

Retrieving all “href” attributes means that the crawler might add garbage link zip-files and non-HTML files to the queue. A solution could be to look at the “HTTP Response Header” in order to determine the content type of the response. Looking for a “.html” file extension in the link would exclude too many proper links, e.g. http://domain.com/news/1

The pages are added to the queue breadth-first.

```
public static List<Uri> GetLinks(Uri baseUri, string text)
{
    var links = new List<Uri>();

    var matches = Regex.Matches(text, @"href=\""(.*?)\""");

    foreach (Match match in matches)
    {
        string href = match.Groups[1].Value.ToLower();

        if (Uri.IsWellFormedUriString(href, UriKind.Relative))
        {
            Uri link = new Uri(baseUri, href);
            links.Add(link);
        }
        else if (Uri.IsWellFormedUriString(href, UriKind.Absolute))
        {
            Uri link = new Uri(href);
            links.Add(link);
        }
    }

    return links;
}
```

## Politeness using robots.txt

To comply with the permissions defined by the owner (robots.txt) is downloaded and a set of restrictions per “User-agent” is stored in the data structure “Site”.

```
public static List<Uri> GetAllowedLinks(Site site, Uri baseUri, string text)
{
    var links = GetLinks(baseUri, text);

    if (site.HasAgent("*"))
    {
        var restrictions = site.GetRestrictions("*");

        foreach (var disallow in restrictions.Disallows)
        {
            // Remove all links relative to the disallow path
            ...
        }
    }

    return links;
}
```

## Removing duplicates and near-duplicates

I did not implement this part in crawler. However, I did implement a Jaccard similarity algorithm to retrieve shingles from a string.

```
public static double GetJaccardSimilarity(string a, string b, int k)
{
    var shinglesA = GetShingles(a, k);
    var shinglesB = GetShingles(b, k);

    double overlap = shinglesA.Intersect(shinglesB).Count();
    double union = shinglesA.Union(shinglesB).Count();

    return overlap / union;
}

private static List<int> GetShingles(string text, int k)
{
    List<int> hashes = new List<int>();

    string[] words = text.Split(null);

    for (int i = 0; i < words.Count() - k; i++)
    {
        string shingle = "";

        for (int j = i; j < i + k; j++)
            shingle += words[j] + " ";

        int hash = shingle.GetHashCode();

        hashes.Add(hash);
    }

    return hashes;
}
```

# Web Intelligence

Mini Project 1 - Indexer

This indexer assumes that all pages are valid HTML files. I am using the .NET extension HtmlAgilityPack, which features convenient methods for traversing an HTML document. All text within HTML tags is added to the list of tokens. However, some HTML tags are not meant for storing text, e.g. “script”, “style”.

These tags are handled using a predefined blacklist of tags that are removed from the document before it is traversed.

```
private static List<string> tagsBlacklist = new List<string>() { "script", "style" };

public static List<string> GetTokens(string html)
{
    HtmlDocument doc = new HtmlDocument();
    doc.LoadHtml(html);

    doc.DocumentNode.Descendants()
        .Where(n => tagsBlacklist.Contains(n.Name))
        .ToList()
        .ForEach(n => n.Remove());

    var chunks = new List<string>();

    foreach (var item in doc.DocumentNode.DescendantsAndSelf())
    {
        if (item.NodeType == HtmlNodeType.Text)
        {
            if (item.InnerText.Trim() != "")
            {
                chunks.Add(item.InnerText.Trim());
            }
        }
    }

    // Trimming, case-folding and splitting chunks into tokens
    var tokens = (from c in chunks
                  from t in c.Split(' ')
                  select t.ToLower().Trim()).ToList();

    return tokens;
}
```

## Trimming and case folding

All tokens are trimmed for whitespaces and converted into lower-case.

## Stop words

I am using a static list of English stop words from http://www.ranks.nl/stopwords to remove these tokens from the index. Normally you would retrieve the language from the document’s “html lang attribute” or analyze the text and determine the language. See implementation in “Frequency Index” section.

## Symbols

All symbols are removed from the tokens using a regular expression.

```
// Removing symbols
for (int i = 0; i < tokens.Count; i++)
    tokens[i] = Regex.Replace(tokens[i], @"[^\w\.@-]", "", RegexOptions.None);
```

## Stemming

Not implemented yet. I could have used Porter's Stemming Algorithm. The following tokens could be represented using one token, namely "connect": "connect", "connector", "connection", "connections", "connected", "connecting".

## Frequency Index

For each page crawled I am creating a frequency index with tokens and their frequency. Tokens are stored in a dictionary with the token as key and frequency as value.

```
public static Dictionary<string, int> GetFrequencyIndex(string html)
{
    var tokens = GetTokens(html);

    // Removing stop words
    tokens = tokens.Except(Lists.StopWords).ToList();

    // Generating the frequency index
    var frequencyIndex = (from t in tokens
                          group t by t into g
                          select g).ToDictionary(p => p.Key, p => p.Count());

    return frequencyIndex;
}
```

## Inverted Index (Boolean)

The inverted index is generated using each page's frequency index.

```
public static Dictionary<string, List<Page>> GetInvertedIndex(List<Page> pages)
{
    var invertedIndex = new Dictionary<string, List<Page>>();

    foreach (var page in pages)
    {
        if (page.FrequencyIndex == null)
            continue;

        foreach (string token in page.FrequencyIndex.Keys)
        {
            // Adding the token if it does not exist in the inverted index
            if (!invertedIndex.ContainsKey(token))
            {
                invertedIndex.Add(token, new List<Page>());
            }

            // A reference to the page is added to the token
            invertedIndex[token].Add(page);
        }
    }

    return invertedIndex;
}
```

# Web Intelligence

## Mini Project 1 – Ranker (content based)

The implementation of the inverted index described in the “Indexer part” is extended to hold information about term frequencies, document frequency and total number of pages in the inverted index. This is achieved by introducing a two new data structures, namely “Index” and “Inverted Index”. The total number of pages and document frequencies are calculated when requested meaning that they are always up-to-date but is inefficient if not cached.

**Term frequency:** Number of occurrences of a given term in a given page
**Document frequency:** Number of pages for a given term
**Total pages:** The distinct number of pages in the inverted index

```
public class InvertedIndex
{
    Dictionary<string, Index> terms = new Dictionary<string, Index>();

    public Dictionary<string, Index> Terms
    {
        get { return terms; }
        set { terms = value; }
    }

    public int TotalPages
    {
        get
        {
            return (from t in terms
                    from p in t.Value.Pages
                    select p).Distinct().Count();
        }
    }
}

public class Index
{
    public int DocumentFrequency
    {
        get { return Pages.Count; }
    }

    // A dictionary of pages and the term frequency
    private Dictionary<Page, int> pages = new Dictionary<Page, int>();

    public Dictionary<Page, int> Pages
    {
        get { return pages; }
    }
}
```

## Ranking

Terms on pages are ranked using the “tf-idf” weighting. Short for “term frequency – inversed document frequency”. It weights a term relative to the occurrences in the inversed index.

This also means that stop words are automatically discarded because they are represented in the most pages. If I was to implement the indexer again I would not remove the stop words, because it eliminates the opportunity to search for stop words.

Three suggested stop words by ranks.nl: “the”, “after” and “under”.

Removing the stop words when indexing will probably generate faulty results on the following queries “the day after tomorrow” and “under wear”.

```
private double GetWeight(Page page, string term)
{
    int totalPages = invertedIndex.TotalPages;

    int termFrequency = page.FrequencyIndex[term];
    int documentFrequency = invertedIndex.Terms[term].DocumentFrequency;

    return Math.Log10(1 + termFrequency) * Math.Log10(totalPages / documentFrequency);
}
```

## Querying

A query is a list of terms that are “normalized” with the same rules as the indexed terms. My implementation does not apply any of these normalizations so it is up to the user to ensure that terms are normalized.

```
public class Query
{
    private List<string> terms = new List<string>();

    public List<string> Terms
    {
        get { return terms; }
    }
}
```

The tf-idf weight is calculated for each term in a query and summed to score a page. The weight is calculated when it is needed which saves space but is very inefficient because it is recalculated for every query.

```
public List<QueryMatch> GetMatches(Query query)
{
    var matches = new List<QueryMatch>();

    foreach (string term in query.Terms)
    {
        // Skip the calculation if the term does not exist in the inverted index
        if (!invertedIndex.Terms.ContainsKey(term))
            continue;

        var pairs = invertedIndex.Terms[term].Pages;

        foreach (var pair in pairs)
        {
            var page = pair.Key;
            double weight = GetWeight(page, term);

            var match = (from m in matches
                         where m.Page == page
                         select m).SingleOrDefault();

            if (match == null)
            {
                match = new QueryMatch();
                match.Page = page;
                match.Score = 0;
                matches.Add(match);
            }

            match.Score += weight;
        }
    }

    return matches;
}

public List<QueryMatch> GetMatches(Query query, int k)
{
    var matches = GetMatches(query);

    return (from m in matches
            orderby m.Score descending
            select m).Take(k).ToList();
}
```