Web Information Retrieval

Lecture 10
Crawling and Near-Duplicate Document Detection
Today’s lecture

- Crawling
- Duplicate and near-duplicate document detection
Basic crawler operation

- Begin with known “seed” pages
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat
Crawling picture

Web

Unseen Web

Seed pages

URLs crawled and parsed

URLs frontier
Simple picture – complications

- Web crawling isn’t feasible with one machine
  - All of the above steps distributed
- Malicious pages
  - Spam pages
  - Spider traps – incl dynamically generated
- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters’ stipulations
    - How “deep” should you crawl a site’s URL hierarchy?
    - Site mirrors and duplicate pages
- Politeness – don’t hit a server too often
What any crawler *must* do

- **Be Polite**: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - Respect *robots.txt* (more on this shortly)
- **Be Robust**: Be immune to spider traps and other malicious behavior from web servers
What any crawler *should* do

- Be capable of **distributed** operation: designed to run on multiple distributed machines
- Be **scalable**: designed to increase the crawl rate by adding more machines
- **Performance/efficiency**: permit full use of available processing and network resources
- Fetch pages of “higher **quality**” first
- **Continuous** operation: Continue fetching fresh copies of a previously fetched page
- **Extensible**: Adapt to new data formats, protocols
Updated crawling picture

- Seed Pages
- URLs crawled and parsed
- URL frontier
- Crawling thread

Unseen Web
URL frontier

- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must try to keep all crawling threads busy
Explicit and implicit politeness

- **Explicit politeness**: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- **Implicit politeness**: even with no specification, avoid hitting any site too often
Robots.txt

- Protocol for giving spiders ("robots") limited access to a website, originally from 1994
  - [www.robotstxt.org/wc/norobots.html](http://www.robotstxt.org/wc/norobots.html)
- Website announces its request on what can(not) be crawled
  - For a URL, create a file `URL/robots.txt`
  - This file specifies access restrictions
Robots.txt example

- No robot should visit any URL starting with "/yoursite/temp/", except the robot called "searchengine":

  User-agent: *
  Disallow: /yoursite/temp/

  User-agent: searchengine
  Disallow:
Processing steps in crawling

- Pick a URL from the frontier
- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

E.g., only crawl .edu, obey robots.txt, etc.
Basic crawl architecture
DNS (Domain Name Server)

- A lookup service on the internet
  - Given a URL, retrieve its IP address
  - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are **blocking**: only one outstanding request at a time
- Solutions
  - DNS caching
  - Batch DNS resolver – collects requests and sends them out together
When a fetched document is parsed, some of the extracted links are *relative* URLs.


During parsing, must normalize (expand) such relative URLs.
Content seen?

- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles
Filters and robots.txt

- **Filters** – regular expressions for URL’s to be crawled/not
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
  - Doing so burns bandwidth, hits web server
- Cache robots.txt files
Duplicate URL elimination

- For a non-continuous (one-shot) crawl, test to see if an extracted+filtered URL has already been passed to the frontier

- For a continuous crawl – see details of frontier implementation
Distributing the crawler

- Run multiple crawl threads, under different processes – potentially at different nodes
  - Geographically distributed nodes
- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate?
Communication between nodes

- The output of the URL filter at each node is sent to the Duplicate URL Eliminator at all nodes.
URL frontier: two main considerations

- **Politeness**: do not hit a web server too frequently
- **Freshness**: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict each other.
(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)
Politeness – challenges

- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is \(>>\) time for most recent fetch from that host
URL frontier: Mercator scheme

URLs

Prioritizer

K front queues

Biased front queue selector
Back queue router

B back queues
Single host on each

Back queue selector

Crawl thread requesting URL
Mercator URL frontier

- URLs flow in from the top into the frontier
- Front queues manage prioritization
- Back queues enforce politeness
- Each queue is FIFO
Front queues

Prioritizer

Biased front queue selector
Back queue router
Front queues

- Prioritizer assigns to URL an integer priority between 1 and $K$
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., “crawl news sites more often”)
Biased front queue selector

- When a back queue requests a URL (in a sequence to be described): picks a front queue from which to pull a URL
- This choice can be round robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized
Back queues

Biased front queue selector
Back queue router

Back queue selector

Heap
Back queue invariants

- Each back queue is kept non-empty while the crawl is in progress
- Each back queue only contains URLs from a single host
  - Maintain a table from hosts to back queues

<table>
<thead>
<tr>
<th>Host name</th>
<th>Back queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.uniroma1.it">www.uniroma1.it</a></td>
<td>3</td>
</tr>
<tr>
<td><a href="http://www.cnn.com">www.cnn.com</a></td>
<td>27</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Back queue heap

- One entry for each back queue
- The entry is the earliest time $t_e$ at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
  - Last access to that host
  - Any time buffer heuristic we choose
Back queue processing

- A crawler thread seeking a URL to crawl:
- Extracts the root of the heap
- Fetches URL at head of corresponding back queue $q$ (look up from table)
- Checks if queue $q$ is now empty – if so, pulls a URL $\nu$ from front queues
  - If there’s already a back queue for $\nu$’s host, append $\nu$ to $q$ and pull another URL from front queues, repeat
  - Else add $\nu$ to $q$
- When $q$ is non-empty, create heap entry for it
Number of back queues $B$

- Keep all threads busy while respecting politeness
- Mercator recommendation: three times as many back queues as crawler threads
Duplicate/Near-duplicate detection

- **Duplication**: Exact match with fingerprints
- **Near-Duplication**: Approximate match
  - **Overview**
    - Compute syntactic similarity with an edit-distance measure
    - Use similarity threshold to detect near-duplicates
      - E.g., Similarity > 80% => Documents are “near duplicates”
      - Not transitive though sometimes used transitively
Duplicate documents

- The web is full of duplicated content
- Strict duplicate detection = exact match
  - Not as common
- But many, many cases of near duplicates
  - E.g., last-modified date the only difference between two copies of a page
Computing near similarity

- Features:
  - Segments of a document (natural or artificial breakpoints)
  - Shingles (Word N-Grams) [Brod98]
    
    "a rose is a rose is a rose" =>

    a_rose_is_a
    rose_is_a_rose
    is_a_rose_is
    a_rose_is_a

- Similarity Measure
  - TFIDF
  - Set intersection
    
    (Specifically, Size_of_Intersection / Size_of_Union )
Computing near similarity

- **Features:**
  - Segments of a document (natural or artificial breakpoints)
  - *Shingles* (Word N-Grams) [Brod98]
    
    “a rose is a rose is a rose” =>
    
    a_rose_is_a
    
    rose_is_a_rose
    
    is_a_rose_is
    
    a_rose_is_a

- **Similarity Measure**
  - TFIDF
  - Set intersection
    
    (Specifically, Size_of_Intersection / Size_of_Union)

\[
\text{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}
\]
Shingles + Set intersection

- Computing **exact** set intersection of shingles between all pairs of documents is expensive/intractable
  - Approximate using a cleverly chosen subset of shingles from each (a **sketch**)
- Estimate **Jaccard** based on a short sketch

![Diagram](Diagram)
Shingles + Set intersection

- Computing **exact** set intersection of shingles between all pairs of documents is expensive and infeasible
- Approximate using a cleverly chosen subset of shingles from each (a **sketch**)


Shingles + Set intersection

- Estimate **Jaccard** based on a short sketch
- Create a “sketch vector” (e.g., of size 200) for each document
  - Documents which share more than $t$ (say 80%) corresponding vector elements are **similar**
  - For doc $D$, sketch$[i]$ is computed as follows:
    - Let $f$ map all shingles in the universe to $0..2^m$ (e.g., $f = $ fingerprinting)
    - Let $\pi_i$ be a specific random permutation on $0..2^m$
    - Pick $\text{sketch}[i] := \text{MIN} \{\pi_i (f(s))\}$ over all shingles $s$ in $D$
Computing Sketch[i] for Doc1

Document 1

Start with 64 bit shingles

Permute on the number line with $\pi_i$

Pick the min value
Computing Sketch[i] for Doc1

Start with 64 bit shingles

Permute on the number line with $\pi_i$

Pick the min value
Test if Doc1.Sketch[i] = Doc2.Sketch[i]

Are these equal?

Test for 200 random permutations: $\pi_1, \pi_2, \ldots, \pi_{200}$
However...

A = B iff the shingle with the MIN value in the union of Doc1 and Doc2 is common to both (i.e., lies in the intersection)

This happens with probability:

\[
\frac{\text{Size of intersection}}{\text{Size of union}}
\]
Set Similarity of sets X, Y

View sets as columns of a matrix M; one row for each element in the universe. $m_{ij} = 1$ indicates presence of item i in set j

Example

$Jaccard(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

$Jaccard(X,Y) = \frac{2}{5} = 0.4$
Key Observation

- For columns $C_i$, $C_j$, four types of rows
  
  \[
  \begin{array}{ccc}
  & X & Y \\
  A & 1 & 1 \\
  B & 1 & 0 \\
  C & 0 & 1 \\
  D & 0 & 0 \\
  \end{array}
  \]

- Overload notation: $A = \#$ of rows of type A

- Claim

\[
\text{Jaccard}(X, Y) = \frac{A}{A + B + C}
\]
“Min” Hashing

- Randomly permute rows
- Hash \( h(X) = \) index of first row with 1 in column \( X \)
- Surprising Property

\[
P\left( h(X) = h(Y) \right) = Jaccard(X, Y)
\]

- Why?
  - Both are \( A/(A+B+C) \)
  - Look down columns \( X, Y \) until first non-Type-D row
  - \( h(X) = h(Y) \) \( \Leftrightarrow \) type A row
Min-Hash sketches

- Pick \( P \) random row permutations
- MinHash sketch
  \[ \text{Sketch}_D = \text{list of } P \text{ indexes of first rows with 1 in column } C \]

- Similarity of signatures
  - Let \( \text{sim}[	ext{sketch}(X),\text{sketch}(Y)] = \text{fraction of permutations where MinHash values agree} \)
  - Observe \( E[\text{sim}(	ext{sketch}(X),\text{sketch}(Y))] = \text{Jaccard}(X,Y) \)
Question

- Document $D_1 = D_2$ iff $\text{size}_{\text{of}} \text{intersection} = \text{size}_{\text{of}} \text{union}$?
### Example

<table>
<thead>
<tr>
<th>C₁</th>
<th>C₂</th>
<th>C₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>R₂</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>R₃</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>R₄</td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>R₅</td>
<td>0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

#### Signatures

<table>
<thead>
<tr>
<th></th>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perm 1 = (12345)</td>
<td>1 2 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perm 2 = (54321)</td>
<td>4 5 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perm 3 = (34512)</td>
<td>3 5 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Similarities

<table>
<thead>
<tr>
<th></th>
<th>1-2</th>
<th>1-3</th>
<th>2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col-Col</td>
<td>0.00</td>
<td>0.50</td>
<td>0.25</td>
</tr>
<tr>
<td>Sig-Sig</td>
<td>0.00</td>
<td>0.67</td>
<td>0.00</td>
</tr>
</tbody>
</table>
All signature pairs

- Now we have an extremely efficient method for estimating a Jaccard coefficient for a single pair of documents.
- But we still have to estimate $N^2$ coefficients where $N$ is the number of web pages.
  - Still slow
- One solution: locality sensitive hashing (LSH)
- Another solution: sorting (Henzinger 2006)
Resources

- IIR Chapters 20, 19.6