Introduction to
Information Retrieval

CS276
Information Retrieval and Web Search
Chris Manning and Pandu Nayak
Crawling and Duplicates
Today’s lecture

- Web Crawling
- (Near) duplicate detection
Basic crawler operation

- Begin with known “seed” URLs
- 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

Seed pages

URLs crawled and parsed

Unseen Web

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 **Robust**: Be immune to spider traps and other malicious behavior from web servers

- Be **Polite**: Respect implicit and explicit politeness considerations
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/robotstxt.html](http://www.robotstxt.org/robotstxt.html)
- Website announces its request on what can(not) be crawled
  - For a server, create a file `/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:
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
What any crawler *should* do

- 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
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)
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
Parsing: URL normalization

- When a fetched document is parsed, some of the extracted links are *relative* URLs.
- E.g., `http://en.wikipedia.org/wiki/Main_Page` has a relative link to `/wiki/Wikipedia:General_disclaimer` which is the same as the absolute URL `http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer`.
- 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
  - Second part of this lecture
Filters and robots.txt

- **Filters** – regular expressions for URLs 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 and share URLs?
Communication between nodes

- Output of the URL filter at each node is sent to the Dup URL Eliminator of the appropriate node.
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 with 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

1

K
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”)

Sec. 20.2.3
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>...</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B</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 $v$ from front queues
  - If there’s already a back queue for $v$’s host, append $v$ to it and pull another URL from front queues, repeat
  - Else add $v$ 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
Introduction to Information Retrieval

Near duplicate document detection
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
Duplicate/Near-Duplicate Detection

- **Duplication**: Exact match can be detected 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
Computing Similarity

- Features:
  - Segments of a document (natural or artificial breakpoints)
  - Shingles (Word N-Grams)
  - \textit{a rose is a rose is a rose} $\rightarrow$ 4-grams are
    - \texttt{a\_rose\_is\_a}
    - \texttt{rose\_is\_a\_rose}
    - \texttt{is\_a\_rose\_is}
    - \texttt{a\_rose\_is\_a}

- Similarity Measure between two docs (= sets of shingles)
  - Jaccard coefficient: \(\frac{\text{Size of Intersection}}{\text{Size of Union}}\)
### Shingles + Set Intersection

- Computing **exact** set intersection of shingles between **all** pairs of documents is expensive.

- Approximate using a cleverly chosen subset of shingles from each (a *sketch*).
- Estimate \( \frac{\text{size_of_intersection}}{\text{size_of_union}} \) based on a short sketch.

![Diagram](image-url)

- Document A
- Shingle set A
- Sketch A
- Document B
- Shingle set B
- Sketch B

Jaccard
Sketch of a document

- Create a “sketch vector” (of size \( \sim 200 \)) for each document
  - Documents that share \( \geq t \) (say 80\%) corresponding vector elements are deemed near duplicates
- For doc \( D \), sketch \( D[i] \) is as follows:
  - Let \( f \) map all shingles in the universe to \( 1..2^m \) (e.g., \( f = \text{fingerprinting} \))
  - Let \( \pi_i \) be a random permutation on \( 1..2^m \)
  - Pick \( \text{MIN} \{\pi_i(f(s))\} \) over all shingles \( s \) in \( D \)
Computing Sketch[i] for Doc1

**Document 1**

- Start with 64-bit $f(\text{shingles})$
- 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)

Claim: This happens with probability

\[
\frac{\text{Size_of_intersection}}{\text{Size_of_union}}
\]
Set Similarity of sets $C_i, C_j$

$$Jaccard(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$$

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

- Example

<table>
<thead>
<tr>
<th></th>
<th>$C_1$</th>
<th>$C_2$</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>
</tbody>
</table>

$$Jaccard(C_1, C_2) = \frac{2}{5} = 0.4$$
Key Observation

- For columns $C_i$, $C_j$, four types of rows
  
<table>
<thead>
<tr>
<th>$C_i$</th>
<th>$C_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Claim

$$\text{Jaccard}(C_i, C_j) = \frac{A}{A + B + C}$$
“Min” Hashing

- Randomly permute rows
- Hash $h(C_i) = \text{index of first row with 1 in column } C_i$
- Surprising Property
  \[ P\left[ h(C_i) = h(C_j) \right] = \text{Jaccard}(C_i, C_j) \]
- Why?
  - Both are $A/(A+B+C)$
  - Look down columns $C_i, C_j$ until first non-Type-D row
  - $h(C_i) = h(C_j) \iff \text{type A row}$
Random permutations

- Random permutations are expensive to compute
- Linear permutations work well in practice
  - For a large prime $p$, consider permutations over $\{0, \ldots, p - 1\}$ drawn from the set:
  \[
  \mathcal{F}_p = \{\pi_{a,b} : 1 \leq a \leq p - 1, 0 \leq b \leq p - 1\} \text{ where}
  \]
  \[
  \pi_{a,b}(x) = ax + b \mod p
  \]
Final notes

- Shingling is a *randomized algorithm*
  - Our analysis did not presume any probability model on the inputs
  - It will give us the right (wrong) answer with some probability on *any input*

- We’ve described how to detect near duplication in a pair of documents

- In “real life” we’ll have to concurrently look at many pairs
  - See text book for details