Web Systems and Algorithms
Web Crawling

Chris Brooks

Department of Computer Science
University of San Francisco
Web Crawling

Why would we want a crawler?
Web Crawling

Why would we want a crawler?
- Populate a search engine
- Mirror or archive a site
- Test navigation/reachability
- Look for security vulnerabilities
- Study Web structure
Basic Crawling

The basic crawler is very simple:

queue = [startURL]
while not done:
    newURL = queue.dequeue()
    page = fetch newURL
    for link in page.URLs:
        queue.enqueue(link)
What issues are not addressed here?
Issues

- malformed HTML
- timeouts and server errors
- efficiency
- courtesy
- spider traps
- other content types
- dynamic content
- duplicate pages
Life Cycle of a Page Fetch

- What happens when a page is fetched?
  - DNS resolution of hostname
  - HTTP request to server
  - Parsing of result

- Much of the delay is caused by the first two points.
Digression - how does DNS work?

- DNS is a distributed, hierarchical database.
- Large number of servers worldwide.
- No database contains all DNS entries.
- When a DNS request is issued, a lookup happens on the network. The result is then cached locally.
DNS/crawler mismatch

- Crawlers exhibit low *spatial locality* in cache fetching.
  - Many different hosts are referenced sequentially
- Cache expiration may be too high
- Many DNS implementations cannot handle multiple concurrent lookups.
- Custom DNS component can improve performance
  - Prefetching is also very useful; resolve hosts as soon as a page is parsed
Building an industrial-strength crawler
In building a crawler, it’s important to keep two points of courtesy in mind

- Don’t overload a single server with requests
  - This can be done by randomly reordering URLs in the queue
- Don’t crawl where you are not allowed
  - This is governed by the Robot Exclusion Protocol
Robots.txt

Robots.txt is a file found at the base URL of a website.

It is a set of instructions as to what paths within the site should not be crawled.

Compliance is voluntary.

User-agent: *
Disallow: /search
Disallow: /groups
Disallow: /images
Disallow: /catalogs
Disallow: /catalogues
Disallow: /news
Allow: /news/directory
Fetching in parallel

- Since most of the time in a page fetch is spent waiting for an HTTP request to return, we would like to parallelize these fetches.
- It might be tempting to fork a single thread for each page fetch.
- This turns out not to work very well. Why?
Fetching in parallel

- Since most of the time in a page fetch is spent waiting for an HTTP request to return, we would like to parallelize these fetches.
- It might be tempting to fork a single thread for each page fetch.
- This turns out not to work very well. Why?
  - Overhead of context switching
  - Need to manage mutexes and locks
  - Still using a poll-driven approach
  - Writes to disk may not be ordered well.
  - Threads may not provide the parallelism we want. (e.g. in Python)
Fetching in parallel

- A non-blocking socket makes much more sense here.
  - `select()` is the most common method for this.
- The client calls `read()`, which returns immediately.
- A signal is generated when data is available - this triggers an interrupt.
- The same socket may `multiplex` different connections.
- Pages are then passed on to a separate component for parsing and storage.
Link extraction and normalization

- One a page is fetched, URLs must be extracted and normalized.
- In standard HTML, extraction is straightforward.
- URLs must then be normalized
  - Canonical hostnames used (e.g. www.cs.usfca.edu instead of cs.usfca.edu)
  - Virtual hosting accounted for (multiple sites at same IP)
  - Relative paths fixed.
Eliminating previously visited URLs

Before adding URLs to a work queue, we must eliminate ones that have been visited.

- Note: this is different than duplicate pages. Why?

Typically, URLs are hashed using MD5. (32-128 bits)

Hostname and path are hashed separately to exploit locality when looking up hash values.
Handling malformed HTML

- Pages “in the wild” can contain a wide variety of quirks and errors
  - Malformed HTML, non-Unicode characters, other parser errors.

- Generated pages can also contain “spider traps”
  - http://www.cs.usfca.edu/a/b/a/b/a/b/a/b/a/b/...

- May be intentional (to thwart/discourage crawlers) or unintentional (for example, calendars with a link to the next month).

- In general, spider traps cannot be completely avoided, but may appear in robots.txt
Detecting duplicate pages

- We would also like to detect pages that have already been crawled, so as to avoid duplicating work.
- Why can we not just use URLs?
- We might just hash page content the way that we did URLs.
- What is the problem with this?
What we would like to do is detect *near-duplicates*. On a small scale, we could use *edit distance*. This is the number of operations needed to transform one string into another. Not practical for pairwise comparison of large Web collections.

We really just need to tell whether two pages share a large fraction of their content.
Shingling

- A shingle is a subsequence of tokens in a larger document.

- \( S(d, w) \) denotes the set of all shingles of length \( w \) in document \( d \). (For fixed \( w \), we write \( S(d) \).)

- The **Jaccard coefficient** is a measure of the similarity of two documents. It is defined as:

\[
\frac{|S(d_1) \cap S(d_2)|}{|S(d_1) \cup S(d_2)|}
\]

- If this exceeds a threshold, the documents are assumed to be duplicates.

- We must also pick a \( w \); in practice, \( w = 10 \) is popular.

- We can use hashing to compute Jaccard efficiently; see the Manning text for details.
Work Queues

- Most web servers limit the number of requests per second from a given domain to avoid DoS attacks.
- Our crawler accounts for this by keeping a queue of requests for each server.
- Requests are removed at the maximum possible rate.
Repository

- One a page is parsed and links extracted, it is stored.
- Typically, it is processed in a way that makes it amenable to search
  - This is Wednesday’s lecture
- Both metadata and page contents must be stored.
- To store pages efficiently, two issues must be considered
  - Space can be saved by compressing
  - Compressing single pages can lead to file block fragmentation
Revisiting crawled pages

- Many web pages change frequently, and a search engine will want to reflect this.
- The HTTP protocol contains an 'if-modified-since' parameter that will only fetch recently-visited URLs.
- We would also like some sort of model that would help us predict frequently-changing pages.
- We can base this on the frequency of past updates.
Focused crawling

In some cases, we might only want to collect a subset of the web:

- Pages within a specific domain
- Pages in a particular language
- Pages with some specific content

This is known as *focused crawling*.
Focused crawling

In order to do a topic-specific crawl, we need a few things:

- A model of what pages belonging to our topic look like
- A way to determine whether a given page belongs to a topic
- A way to predict whether a link is likely to lead to a useful page.

We might be more or less aggressive in following unknown links.
Dealing with Web 2.0

“Web 2.0” pages present new challenges
- Links may be fetched, changed, or added dynamically based on application state

For Javascript, code can be executed in a breadth-first fashion; events are triggered and the DOM is analysed for new URLs.

For Flash, this is more challenging, as Flash is not open
- Adobe has partnered with Google to allow them to index .swf files and extract text and links