Magnitude of the crawling problem

To fetch 1,000,000,000 pages in one month . . .
. . . we need to fetch almost 400 pages per second!
Actually: many more since many of the pages we attempt to crawl will be duplicates, unfetchable, spam etc.

Basic crawler operation

- Initialize queue with URLs of known seed pages
- Repeat
  - Take URL from queue
  - Fetch and parse page
  - Extract URLs from page
  - Add URLs to queue
- Fundamental assumption: The web is well linked.

Complications in crawling

- We need many machines — how do we distribute?
- Latency/bandwidth
- How deep should we crawl sites?
- Duplicates
- Spam and spider traps
- Politeness — don’t hit a server too often
Spider trap

- Malicious server that generates an infinite sequence of linked pages
- Sophisticated spider traps generate pages that are not easily identified as dynamic.

What a crawler must do

- Be polite
  - Don't hit each site too often
  - Only crawl pages you are allowed to crawl: robots.txt
- Be robust
  - Be immune to spider traps, duplicates, very large pages, very large websites, dynamic pages etc

Robots.txt

- Protocol for giving crawlers ("robots") limited access to a website, originally from 1994
- Examples:
  - User-agent: *
    Disallow: /yoursite/temp/
  - User-agent: searchengine
    Disallow:
- Important: cache the robots.txt file of each site we are crawling

What any crawler should do

- Be capable of distributed operation
- Be scalable: need to be able to increase crawl rate by adding more machines
- Fetch pages of higher quality first
- Continuous operation: get fresh version of already crawled pages
- Pick a URL from the frontier
- Fetch the document at the URL
- Check if the document has content already seen (if yes: skip following steps)
- Index document
- Parse the document and extract URLs to other docs
- For each extracted URL:
  - Does it fail certain tests (e.g., spam)? Yes: skip
  - Already in the frontier? Yes: skip
**URL normalization**

- Some URLs extracted from a document are **relative** URLs.
- E.g., at http://mit.edu, we may have /aboutsite.html
  - This is the same as: http://mit.edu/aboutsite.html
- During parsing, we must normalize (expand) all relative URLs.

**Content seen**

- For each page fetched: check if the content is already in the index
- Check this using document fingerprints or **shingles**
- Skip documents whose content has already been indexed

**Distributing the crawler**

- Run multiple crawl threads, potentially at different nodes
  - Usually geographically distributed nodes
- Partition hosts being crawled into nodes

**Google data centers:**

http://www.wayfaring.com/maps/show/48030
URL frontier: Two main considerations

- **Politeness**: Don’t hit a web server too frequently
  - E.g., insert a time gap between successive requests to the same server
- **Freshness**: Crawl some pages (e.g., news sites) more often than others
- Not an easy problem: simple priority queue fails.
  - Why?

 Mercator URL frontier

- Prioritizer assigns to URL an integer priority between 1 and $K$.
- Then appends URL to corresponding queue
- Heuristics for assigning priority: refresh rate, PageRank etc

---

Front queues

- URLs flow in from the top into the frontier.
- Front queues manage prioritization.
- Back queues enforce politness.
- Each queue is FIFO.
Biased front queue selector

- Selection from front queues is initiated by back queues (see below)
- Pick a front queue from which to select next URL
  - Round robin
  - Randomly
  - Or more sophisticated variant
  - But with a bias in favor of high-priority front queues

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.

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 by (i) last access to that host (ii) time gap heuristic

How a crawler thread interacts with back queue

- Extract the root of the back queue heap
- Fetch the URL at head of corresponding back queue $q$
- Check if $q$ is now empty
- If yes: keep (i) pulling URLs from the front queues and (ii) adding them to their corresponding back queues until . . .
- . . .the URL’s host does not have a back queue – then put the URL in $q$ and create heap entry for it.
Resources

- Chapter 20 of IIR
- Resources at http://ifnlp.org/ir
- Paper on Mercator by Heydon et al.
- Robot exclusion standard