Introduction to Information Retrieval

Τι θα δούμε σήμερα

1. Web crawlers or spiders (κεφ 20)
2. Personalization/Recommendations
3. Lucene
Spiders (σταχυολόγηση ιστού)

Web crawler or spider

*How hard and why?*

- Getting the content of the documents is easier for many other IR systems.
  - E.g., indexing all files on your hard disk: just do a recursive descent on your file system
- For web IR, getting the content of the documents takes longer, because of latency.
  - But is that really a design/systems challenge?
Βασική λειτουργία

- 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
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
Simple picture – complications

Magnitude of the problem
To fetch 20,000,000,000 pages in one month . . .
we need to fetch almost 8000 pages per second!

- Actually: many more since many of the pages we attempt to crawl will be duplicates, unfetchable, spam etc.

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 (very large pages, very large websites, dynamic pages etc)
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**
- **Unseen Web**
- **URL frontier**
- **Crawling thread**

### 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)

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

E.g., only crawl .edu, obey robots.txt, etc.
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 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: 
  ```
Δημιουργία του σταχυολογητή

DNS (Domain Name Server)

- Ένας lookup υπηρετής στο internet
  - Δίνει ένα URL, προκειμένου να αναζητήσει το IP address
  - Υπηρεσία που διαθέτει μια διανομέα σειρά από κεραίες – οπότε, lookup latencies μπορεί να είναι ρευστά (και για δευτερονόμια)
- Common OS implementations of DNS lookup are blocking: only one outstanding request at a time
- Επιλογές
  - DNS caching
  - Μηχανήματα συγκεντρώνουν αιτήματα και τα στέλνουν τούτα κατ’ αρχήν

Ως δείκτης για αυτή την παράγραφο, χρησιμοποιούμε τον διάγραμμα που παρουσιάζεται στο παράθυρο. Στο διάγραμμα απεικονίζεται η διαδικασία αναζήτησης URL μέσω της διεύθυνσης ιστοσελίδων (URL) μέσω του DNS: από τον ιστότοπο (WWW), υπολογίζεται η διεύθυνση (DNS), παράγεται η ανάγλυφη εικόνα (Content Filter), χρησιμοποιείται η εικόνα (URL Frontier) και τέλος, γίνεται η ελέγχοντας επιπλέον (URL Set).
Introduction to Information Retrieval

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
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
Distributing the crawler

- 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 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
Mercator URL frontier

Goals: ensure that
1. only one connection is open at a time to any host;
2. a waiting time of a few seconds occurs between successive requests
3. high-priority pages are crawled preferentially.

URL frontier: Mercator scheme
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”)
  - Page-rank based

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 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 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 from
  - Last access to that host
  - Any time buffer heuristic we choose

URL processing

A crawler thread seeking a URL to crawl:

- Extracts the root of the heap
- If necessary waits until $t_i$
- Fetches URL at head of corresponding back queue $q$ (look up from table)
URL processing

After fetching the URL

- Checks if (back)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 $q$ 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
How to distribute the term index across a large computer cluster that supports querying.

Two alternatives index implementations
- partitioning by terms or global index organization, and
- partitioning by documents or local index organization.
Index terms partitioned into subsets,
Each subset resides at a node.
Along with the terms at a node, we keep their postings

A query is routed to the nodes corresponding to its query terms.
In principle, this allows greater concurrency since a stream of queries with different query terms would hit different sets of machines.

Documents partitioned into subsets
Each subset resides in a node
Each node contains the index for a subset of all documents.

A query is distributed to all nodes, with the results from various nodes being merged before presentation to the user.
In principle, index partition allows greater concurrency, since a stream of queries with different query terms would hit different sets of machines.

In practice, partitioning indexes by vocabulary terms turns out to be non-trivial.

Multi-word queries require the sending of long postings lists between sets of nodes for merging, and the cost of this can outweigh the greater concurrency.

Load balancing the partition is governed not by an a priori analysis of relative term frequencies, but rather by the distribution of query terms and their co-occurrences, which can drift with time or exhibit sudden bursts.

More difficult implementation.
More common
- trades more local disk seeks for less inter-node communication.
- One difficulty: *global statistics* used in scoring - such as idf –
  - must be computed across the entire document collection even though the index at any single node only contains a subset of the documents.
  - Computed by distributed "background" processes that periodically refresh the node indexes with fresh global statistics.

How to distributed documents to nodes?
- Hash of each URL to nodes

*At query time,*
the query is broadcast to each of the nodes, each node sends each top k results which are merged to find the top k documents for the query
A common implementation heuristic:
Partition the document collection into
- indexes of documents that are more likely to score highly on most queries and
- low-scoring indexes with the remaining documents

Only search the low-scoring indexes when there are too few matches in the high-scoring indexes
Connectivity Server

- Support for fast queries on the web graph
  - Which URLs point to a given URL?
  - Which URLs does a given URL point to?

Stores mappings in memory from
  - URL to outlinks, URL to inlinks

Applications
- Crawl control
- Web graph analysis
  - Connectivity, crawl optimization
- Link analysis

Assume that each web page is represented by a unique integer

Maintain an adjacency table: a row for each web page, with the rows ordered by the corresponding integers.

One for pages link to and one for pages linked to by

Focus on the former
Boldi and Vigna 2004

- Webgraph – set of algorithms and a java implementation
- Fundamental goal – maintain node adjacency lists in memory
  - For this, compressing the adjacency lists is the critical component

Adjacency lists

- The set of neighbors of a node
- Assume each URL represented by an integer
- E.g., for a 4 billion page web, need 32 bits per node
- Naively, this demands 64 bits to represent each hyperlink
Adjacency list compression

- Properties exploited in compression:
  - Similarity (between lists)
  - Locality (many links from a page go to “nearby” pages)
  - Use gap encodings in sorted lists

- Many rows have many entries in common. Thus, if we explicitly represent a prototype row for several similar rows, the remainder can be succinctly expressed in terms of the prototypical row.

- By encoding the destination of a link, we can often use small integers and thereby save space.

- Store the offset from the previous entry in the row

Storage

- Boldi/Vigna get down to an average of ~3 bits/link
  - (URL to URL edge)

- How?
Main ideas of Boldi/Vigna

- Consider lexicographically ordered list of all URLs, e.g.,
  - [www.stanford.edu/alchemy](http://www.stanford.edu/alchemy)
  - [www.stanford.edu/biology](http://www.stanford.edu/biology)
  - [www.stanford.edu/biology/plant](http://www.stanford.edu/biology/plant)
  - [www.stanford.edu/biology/plant/copyright](http://www.stanford.edu/biology/plant/copyright)
  - [www.stanford.edu/biology/plant/people](http://www.stanford.edu/biology/plant/people)
  - [www.stanford.edu/chemistry](http://www.stanford.edu/chemistry)

Boldi/Vigna

- Each of these URLs has an adjacency list
- Main idea: due to templates, the adjacency list of a node is similar to one of the 7 preceding URLs in the lexicographic ordering
- Express adjacency list in terms of one of these
- E.g., consider these adjacency lists
  - 1, 2, 4, 8, 16, 32, 64
  - 1, 4, 9, 16, 25, 36, 49, 64
  - 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
  - 1, 4, 8, 16, 32, 36, 49, 64
  - Encode as (-2), remove 9, add 8
  - Why 7?
ΤΕΛΟΣ α’ μέρους 11ου Μαθήματος

Ερωτήσεις?

Χρησιμοποίησαν κάποια υλικά των:
- Pandu Nayak and Prabhakar Raghavan, CS276: Information Retrieval and Web Search (Stanford)
- Hinrich Schütze and Christina Lioma, Stuttgart IIR class