Lecture 11: Web Crawling

1. How the Web Works
2. Web Crawling

Web Resources

Web Resources are uniquely identified by Uniform Resource Identifiers (URIs):

foo://example.com:8042/over/there?name=ferret#nose

- Scheme
- Authority
- Path
- Query
- Fragment

Most common: HTTP, the Hypertext Transfer Protocol

HTTP

Typical HTTP URIs look like this:

http://www.google.com/search?q=ifis
http://en.wikipedia.org/wiki/New_South_Wales#History

The World Wide Web = Resources + hyperlinks
Normalized URLs

- In HTTP, every URI has a normalized form
- Normalization affects:
  - (Un)quoting of special characters (e.g. %7E represents ~)
  - Case normalization (i.e. transform the hostname to lowercase)
  - Remove the default port (HTTP's default port is 80)
  - Remove path segments "." and ".."

  ![HTTP Request](http://abc.com/~smith/home.html)
  ![HTTP Response](http://ABC.com/%7Esmith/home.html)
  ![HTTP Server](http://ABC.com:/%7Esmith/home.html)
  ![HTTP Request](http://abc.com:/~smith/../~smith/home.html)
  ![HTTP Server](http://ABC.com/././~smith/home.html)

HTTP/1.1 200 OK
Cache-Control: private, max-age=0

Some information related to caching
MIME type of this resource
The resource itself

TCP/IP is based on IP addresses
- Therefore: When some client wants to contact the host www.google.com, it has to look up the host’s IP address first.
- How do HTTP requests look like?

HTTP request:
GET /search?q=ifis HTTP/1.1
Host: www.google.com
Connection: close
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Accept-Encoding: gzip
Accept-Language: de,en;q=0.7,en-us;q=0.3
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7

HTTP response
HTTP/1.1 200 OK
Cache-Control: no-store, no-cache, must-revalidate, post-only
Expires: -1
Date: Tue, 27 Jan 2009 10:03:57 GMT
Server: gws
Accept-Language: de,en;q=0.7,en-us;q=0.3
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Connection: close
Transfer-Encoding: chunked

<!doctype html><head><meta http-equiv=content-type content="text/html; charset=UTF-8"><title>ifis - Google Search<title><script></script><style>

Status code (200 means “resource found”)

HTTP is a request/response standard between a client and a server

GET: Requests a representation of the specified resource
HEAD: Asks for the response identical to the one that would correspond to a GET request, but without the response body (useful to determine whether the resource has changed)

Important types of HTTP requests are:
- GET:
- HEAD:
- POST:

HTTP works on top of TCP/IP:
- Servers are identified by IP addresses (e.g. 134.169.32.171)
- Hostnames are mapped to IP addresses using the Domain Name System (DNS)
- There is a many-to-many relationship between IP addresses and hostnames
How Does HTTP Work? (6)

- Important types of HTTP status codes are:
  - 200 (OK): Standard response for successful HTTP requests
  - 301 (Moved Permanently): This and all future requests should be directed to a given URI
  - 302 (Found / Moved Temporarily): Only this request should be directed to a given URI
  - 304 (Not Modified): The resource has not been modified since last requested
  - 404 (Not Found): The requested resource could not be found (but may be available again in the future)
  - 410 (Gone): The requested resource is no longer available (and will not be available again)

What we have learned:
- How Web resources are identified (URIs)
- How Web resources can be retrieved (HTTP)

What's still missing: How do resources look like?

- Most web resources are of MIME type text/html, i.e. they are text documents written using HTML
- HTML stands for Hypertext Markup Language
- HTML was invented by Tim Berners-Lee in 1991

HTML

- What we have learned:
  - How Web resources are identified (URIs)
  - How Web resources can be retrieved (HTTP)

- What's still missing: How do resources look like?

- Most web resources are of MIME type text/html, i.e. they are text documents written using HTML
- HTML stands for Hypertext Markup Language
- HTML was invented by Tim Berners-Lee in 1991

HTML (2)

- HTML is a markup language, i.e., it provides means to describe the structure of text-based information in a document
- In HTML you can denote certain text as...
  - Headings:
    ```html
    <h1>Main heading</h1>  
    <h2>Sub heading</h2>
    ```
  - Paragraphs:
    ```html
    <p>Some text...</p>
    ```
  - Lists:
    ```html
    <ul>
      <li>First item</li>
      <li>Second item</li>
    </ul>
    ```
  - Links:
    ```html
    <a href="http://www.google.com">Link to Google</a>
    ```
  - ...
The Beginnings of the Web

Berners-Lee’s NeXTcube:

- 25 MHz CPU, 8 MB–64 MB RAM

19

The Beginnings of the Web

1990

- CERN computer scientist Robert Cailliau joins Berners-Lee’s vision and rewrites the proposal
- Both present their idea at the European Conference on Hypertext Technology but find no vendors who support them
- The name World Wide Web is born
- By Christmas 1990, all tools for a working Web have been created by Berners-Lee:
  - HTML
  - HTTP
  - A Web server software: CERN httpd
  - A Web server: http://info.cern.ch
  - A Web browser/editor: WorldWideWeb (runs only on NeXT)

The Beginnings of the Web

1991

- Nicola Pellow creates a simple text browser that could run on almost any computer
- To encourage use within CERN, they put the CERN telephone directory on the Web, which previously was located on a mainframe
- Berners-Lee announces the Web in the alt.hypertext newsgroup: “The WorldWideWeb (WWW) project aims to allow all links to be made to any information anywhere. [...] The WWW project was started to allow high energy physicists to share data, news, and documentation. We are very interested in spreading the web to other areas, and having gateway servers for other data. Collaborators welcome!”

The Beginnings of the Web

1993

- The Web spreads around the world
- The graphical Web browser Mosaic is developed by a team at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign; the team is led by the later founder of Netscape, Marc Andreessen

1994

- Netscape is founded
- Mosaic becomes the Netscape Navigator
- The World Wide Web Consortium (W3C) is founded by Berners-Lee at the Massachusetts Institute of Technology with support from the Defense Advanced Research Projects Agency (DARPA) and the European Commission

Lecture 11: Web Crawling

1. How the Web Works
2. Web Crawling
MUST-Have Features

A Basic Crawler

- A basic crawler (aka robot, bot, spider) consists of:
  - A queue of URIs to be visited
  - A method to retrieve Web resources and process HTTP data
  - A page parser to extract links from retrieved resources
  - A connection to the search engine’s index

- The basic mode of operation:
  1. Initialize the queue with URIs of known seed pages
  2. Take URI from queue
  3. Retrieve and parse page
  4. Extract URIs from page
  5. Add new URIs to queue
  6. GOTO (2)

Problem Size

- The Web is large: 60 billion pages (more or less…)
- Let’s assume we want to crawl each page once a year

- How many pages do we have to crawl per second then?
  - 60,000,000,000 pages per year
  - 5,000,000,000 pages per month
  - 166,666,667 pages per day
  - 6,944,444 pages per hour
  - 115,740 pages per minute
  - 1929 pages per second

- Well, it seems like we need a highly scalable crawler...

Further Complications

- Apart from scalability, there are further issues

  - How to detect spam pages?
  - How to detect duplicates or pages already seen?
  - How to avoid spider traps?
  - We need many machines, how do we distribute?
  - How to handle latency problems?
  - How to limit the used bandwidth?
  - How deep should we crawl sites?
  - How to comply with the site owner’s wishes?

MUST-Have Features (2)

- Politeness
  - Web site owner’s usually have to pay for their Web traffic
  - Do not generate unnecessarily high traffic!
  - Do not slow down other people’s servers by “hammering,” i.e., keep the number of requests per time unit low!
  - Obey explicit crawling policies set by site owners (e.g. robots.txt)

- Robustness
  - Golden rule:
    - For every crawling problem you can (or cannot) think of, there will be a Web page exhibiting this problem
    - Web pages, URLs, HTTP responses, and network traffic as such can be malformed and might crash your software
    - Therefore, use very robust software
    - “Very robust” usually means non-standard
  - Robustness also refers to the ability to avoid spider traps

- The robot exclusion standard
  - Exclude some resources from access by robots, and thus from indexing by search engines
  - Put a file named robots.txt in your domain’s top-level directory (e.g. http://en.wikipedia.org/robots.txt), which specifies what resources crawlers are allowed to access
  - Caution: This “standard” is not a standard in the usual sense, it’s purely advisory!

- Examples:
  - Allow all robots to view all files:
    User-agent: *
    Disallow: *
More examples:

- Keep all robots out:
  User-agent: *
  Disallow: /

- Exclude certain resources:
  User-agent: *
  Disallow: /cgi-bin/
  Disallow: /private/

- Exclude a specific bot:
  User-agent: BadBot
  Disallow: /private/

- Limit the number of requests per second:
  Request-rate: 1/5

- Recommend a visit time interval (in GMT):
  Visit-time: 0600-0845

A look at http://www.wikipedia.org/robots.txt:

## robots.txt for http://www.wikipedia.org/ and friends

## Please note: There are a lot of pages on this site, and there are # some misbehaved spiders out there that go _way_ too fast. If you're # irresponsible, your access to the site may be blocked.

# advertising-related bots:
User-agent: Mediapartners-Google*
Disallow: /

# Wikipedia work bots:
User-agent: IsraBot
Disallow: /

# Crawlers that are kind enough to obey, but which we'd rather not have # unless they're feeding search engines.
User-agent: uBirCrawler
Disallow: /

User-agent: DOC
Disallow: /

User-agent: Zao
Disallow: /

# Some bots are known to be trouble, particularly those designed to copy # entire sites. Please obey robots.txt.
User-agent: sitescheck.internetscout.com
Disallow: /

## Sorry, wget in its recursive mode is a frequent problem.
# Please read the man page and use it properly; there is a # --wait option you can use to set the delay between hits,
# for instance.
User-agent: wget
Disallow: /

# The 'grub' distributed client has been *very* poorly behaved.
User-agent: grub-client
Disallow: /

## Doesn't follow robots.txt anyway, but...
User-agent: k2spider
Disallow: /

# Hits many times per second, not acceptable
# http://www.nameprotect.com/botinfo.html
User-agent: WPbot
Disallow: /

# A capture bot, downloads gazillions of pages with no public benefit
# http://www.webreaper.net/
User-agent: WEBReaper
Disallow: /
**SHOULD-Have Features**

- **Distributed:**
  - The crawler should have the ability to execute in a distributed fashion across multiple machines

- **Scalable:**
  - The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth

- **Performance and efficiency:**
  - The crawl system should make efficient use of various system resources including processor, storage, and network bandwidth

- **Quality:**
  - The crawler should be biased towards fetching “useful” pages first and updating them more often than “useless” ones

**Anatomy of a Large-Scale Crawler**

**The DNS Handler**

- Fetching DNS information usually is slow due to network latency and the need to query many servers in parallel
- The DNS handler is a customized local DNS component
  - Prefetches DNS information that will be needed by some work-thread in the near future
  - Uses a relaxed policy regarding DNS updates, i.e., break the DNS standard to avoid unnecessary DNS queries

**The Duplicate URI Checker**

- **Task:**
  - Find out whether a given URI is contained in the URI pool
  - But: As quick as possible!

- **Problems:**
  - Doing string comparisons with all pool URIs is too expensive
  - Even using index structures does not help much here since string operations as such are very expensive

- **Solution:**
  - Use fingerprints!
A B-tree:

- B-trees can be searched efficiently
- Numerical comparisons can be done quickly

The Duplicate URI Checker (3)

The Duplicate URI Checker (4)

The whole process:

- Is the given URI's fingerprint contained in the B-tree?
- Yes
- No
- Do the matching fingerprints originate from the same URI strings?
- Yes
- No
- It's a known URI
- It's a new URI

The Duplicate URI Checker (5)

- Problem size?
  - Let's say we have collected 1 billion URIs
  - Each URI's fingerprint requires at least 16 bytes
  - To store 1 billion URIs, we need about 15 GB of storage
  - Plus much more space to store URI strings and metadata
- There are two options of storage:
  - A distributed main memory index
  - Put it on disk
- In both cases, it would be reasonable to enforce some locality by grouping URIs together that usually will be accessed in quick succession

The Duplicate URI Checker (6)

- How to enforce locality?
  - Observation: URIs having the same hostname are usually accessed together in crawling
  - Idea: Take two fingerprints per URI
    - One for the hostname
    - One for the rest
    - Concatenate both to form a URI's fingerprint
  - Then, URIs of the same hostname are located in the same sub-tree of the index

The Duplicate Content Checker

- In principle, we could check for duplicate content in the same way as we did it for duplicate URIs
- But what about this page?

  ![Image of a clock showing 10:05:14]

  Right now, the official U.S. time is:

  10:05:14
  Tuesday, January 27, 2009

  You choose the Central Timezone
  Government Harmonized Time, Zulu: GMT-6
  Simplified历史性

- Or, think of pages with ads that change on every visit

The Duplicate Content Checker (2)

- This problem is called near-duplicate detection

  ![Image of two pages]

  First step: Focus on content!
    - Remove all styling information from the Web resource
    - Convert the resource into a text-only view
    - Drop textual information like navigation structures
    - Drop images and dynamic content
The Duplicate Content Checker (3)

Example:

The Institute for Information Systems at Technische Universität Braunschweig, Germany, focuses on research and teaching in the area of databases and information systems.

The Duplicate Content Checker (4)

• After this step, the problem amounts to near-duplicate detection on plain text documents (word sequences)
  • It can be solved using a technique called shingling
    – Given: A positive number \( k \) and a sequence of terms \( d \)
    – Definition: The \( k \)-shingles of \( d \) are the set of all consecutive sequences of \( k \) terms in \( d \)
  • Example:
    – \( d = \text{"a rose is a rose is a rose"} \)
    – \( k = 4 \) (a typical value used in the near-duplicate detection of Web pages)
    – The 4-shingles of \( d \) are:
      “a rose is a” “rose is a” “is a rose is”

The Duplicate Content Checker (5)

• Intuitive idea: Two documents are near-duplicates if the two sets of shingles generated from them are nearly the same
  • A more precise definition:
    Let \( d \) and \( d' \) be documents and let \( S(d) \) and \( S(d') \) be their respective sets of shingles
    – Remember the Jaccard coefficient from fuzzy retrieval
    – We use it to measure the overlap between the sets:
      \[ J(S(d), S(d')) = \frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|} \]
  • Define \( d \) and \( d' \) to be near-duplicates if \( J(\ldots) \) is “large,” e.g. larger than 0.9

The Duplicate Content Checker (6)

\[ J(S(d), S(d')) = \frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|} \]

• Computing the value of \( J(S(d), S(d')) \) directly is easy
  • Complexity is \( O(n \log n) \)
    – Sort each set of shingles
    – Find intersection and union by merging the two sorted lists
  • However, the typical situation is different:
    – We already have a large document collection
    – We want to check whether a new document is a near-duplicate
    – Compare the new document with all existing ones
      • Too expensive, we need some clever indexing technique...

The Duplicate Content Checker (7)

• A very clever indexing technique (to be discussed later) relies on a randomized approximation algorithm for computing \( J(S(d), S(d')) \)
  • To explain this algorithm, we need the following:
    – Map every shingle into a hash value over a large space, say the space of all 64-bit integers
    – Let \( H(d) \) be the set of hash values derived from \( S(d) \)
      \[ H(d) \]
        Set of hash values (64-bit integers)
    – Apply hash function to each shingle
    – Then, it is
      \[ J(S(d), S(d')) = J(H(d), H(d')) \]

The Duplicate Content Checker (8)

• Let \( \pi \) be a random permutation on the set of all 64-bit integers, i.e. \( \pi \) is a one-to-one function that maps any 64-bit integer to some 64-bit integer
  • The simplest permutation is the identity mapping
  • Another example of a permutation is \( \pi(x) = (x + 1) \mod 2^m \)
  • Here, “random” means chosen at random according to the uniform distribution over the set of all permutations on the set of all 64-bit integers
  • When applying a single permutation \( \pi \) to each hash value in \( H(d) \), we get a new set of 64-bit numbers \( \pi(H(d)) \)

• Furthermore, let \( \min(\pi(H(d))) \) be the smallest number in \( \pi(H(d)) \)

The Duplicate Content Checker (9)

• The simplest permutation is the identity mapping
• Another example of a permutation is \( \pi(x) = (x + 1) \mod 2^m \)
• Here, “random” means chosen at random according to the uniform distribution over the set of all permutations on the set of all 64-bit integers
• When applying a single permutation \( \pi \) to each hash value in \( H(d) \), we get a new set of 64-bit numbers \( \pi(H(d)) \)

• Furthermore, let \( \min(\pi(H(d))) \) be the smallest number in \( \pi(H(d)) \)
Given: 
- $S(d)$ and $S(d')$: The sets of shingles 
- $H(d)$ and $H(d')$: The sets of corresponding hash values 
- $\wp$: A random permutation on the set of 64-bit integers 
- $\Pi(d)$ and $\Pi(d')$: The result of applying $\wp$ to $H(d)$ and $H(d')$ 
- $\min(\Pi(d))$ and $\min(\Pi(d'))$: The minimum positions of the first "1" columns

Then, the following is true, for any documents $d$ and $d'$:

$$ f(H(d), H(d')) = Pr(\min(\Pi(d)) = \min(\Pi(d'))) $$

Intuitive meaning: The overlap between the sets of shingles (measured by the Jaccard coefficient) is same as the probability that their corresponding hash sets have the same smallest number when permuted randomly.

This identity will allow us to build an indexing schema that supports efficient near-duplicate detection for new documents.

How to prove this identity?

We have to prove the following:

$$ f(H(d), H(d')) = Pr(\min(\Pi(d)) = \min(\Pi(d'))) $$

Proof: 
- First, represent the sets $H(d)$ and $H(d')$ as bit strings of length $2^{64}$, where the $i$-th bit is set if number $i$ is contained in $H(d)$ or $H(d')$, respectively.

Example: $H(d) = \{1, 2, 4, ..., 2^{52} - 2, 2^{52} - 1\}$ $H(d') = \{0, 2, 4, 5, ..., 2^{64} - 1\}$

<table>
<thead>
<tr>
<th>$d'$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>$\ldots$</th>
<th>$2^{52} - 2$</th>
<th>$2^{52} - 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H(d)$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\ldots$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$H(d')$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- The permutation $\wp$ corresponds to a random swapping of columns, resulting in bit strings $\Pi(d)$ and $\Pi(d')$.

<table>
<thead>
<tr>
<th>$d'$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>$\ldots$</th>
<th>$2^{52} - 2$</th>
<th>$2^{52} - 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi(d)$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\ldots$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\Pi(d')$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\ldots$</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• What's $Pr$(the first non-"0–0" column is a "1–1" column)?
- Since $\wp$ is uniformly distributed over all permutations of columns:

$$ Pr(\text{the first non-"0–0" column is a "1–1" column}) = Pr(\min(\Pi(d)) = \min(\Pi(d'))) $$

Proof (continued):
- $Pr(\min(\Pi(d)) = \min(\Pi(d'))) = Pr(\text{the first non-"0–0" column is a "1–1" column})$
- This is exactly the definition of the Jaccard coefficient!
That's great!

- We can estimate the overlap between \( H(d) \) and \( H(d') \) by applying random permutations and comparing the minima
  - Estimate \( \Pr(\min(\Pi(d)) = \min(\Pi(d'))) \) by drawing random samples
  - The literature says that 200 is a good number of random permutations/samples to use in practice

- Therefore, let \( n_1, n_2, \ldots, n_{200} \) be a fixed(!) set of permutations, which has been generated randomly
  - Let \( \nu(d) = (\min(\Pi_1(d)), \min(\Pi_2(d)), \ldots, \min(\Pi_{200}(d))) \)
  - \( \nu(d) \) is called the sketch of \( d \)

- Then, the Jaccard coefficient of \( H(d) \) and \( H(d') \) can be estimated by counting the number of places in which \( \nu(d) \) and \( \nu(d') \) agree

- Since \( J(H(d), H(d')) \) and \( J(\nu(d), \nu(d')) \) usually are very similar, we finally arrived at a method for estimating \( J(\nu(d), \nu(d')) \)

Again, there is a trick:
- For each indexed document \( d \) and each entry \( \nu(d) \) of its sketch
- Consider two documents to be near-duplicates if their sketches have at least \( m \) matching places, we restrict our search to all documents in the B-tree which have at least \( m \) numbers in common

- The set of all indexed documents \( d' \) such that \( \nu(d) \in \nu(d') \)

- Only these documents' sketches can have a non-zero overlap with \( d \)'s sketch

Extension:
- If we consider two documents to be near-duplicates if their sketches have at least \( m \) matching places, we restrict our search to all documents in the B-tree which have at least \( m \) numbers in common

- The set of all these documents can be found by intersecting the sets of documents having at least 1 number in common
**Focused Crawling**

- Now, assume that you own a Web search engine that focuses on a **specific topic**, e.g. sports
  - Then, it would be reasonable to do some kind of “focused crawling” to avoid crawling unrelated pages
- **How to do it?**
  - Train a **classifier** that is able to detect whether a web page is about the relevant topic
  - Start crawling with a hand-crafted set of highly on-topic pages
  - When crawling, only follow out-links of on-topic pages
- **Possible extension:**
  - For any yet unseen page, estimate the probability that this page is on-topic using a clever model
  - Do the crawl in order of descending probabilities

**Crawler-Friendly Web Sites**

- **How to build crawler-friendly web sites?**
  - Use a robots.txt to exclude “uninteresting” content
  - Create a static sitemap
  - Write “good” HTML
  - Avoid scripted content whenever you can
  - Provide caching and freshness information in the HTTP headers
    - [http://www.ircache.net/cgi-bin/cacheability.py](http://www.ircache.net/cgi-bin/cacheability.py)
  - Send correct HTTP status codes
    - In particular: Use standard redirects
  - Send the correct MIME types and content encodings
  - Use canonical hostnames
  - Avoid spider traps (e.g. stupidly designed session IDs)
  - Annotate images (ALT/LONGDESC attribute) if appropriate(!)

**Next Lecture**

- Exploiting the Web graph for ranking
  - HITS
  - PageRank

**Comparison to unfocused crawling:**

- Time allowed for the crawl
- Focused crawling: faster training, higher precision
- Unfocused crawling: faster crawling, higher recall

**Exploiting the Web graph for ranking**

- HITS
- PageRank