Exercise 8.1
What are the major differences between Web search and classical information retrieval?

Exercise 8.2
How does a Web search provider make its money?

Exercise 8.3
What are the major challenges for Web crawlers?

Exercise 8.4
How can you estimate the size of the Web? What are possible problems here?

Exercise 8.5
How does the Web look like?
A typical Web search engine:

1. How the Web Works
2. Web Crawling

The Web

The World Wide Web = Resources + hyperlinks

HTTP

Typical HTTP URIs look like this:

- 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 “..”

Normalized URIs

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

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

- Important types of HTTP requests are:
  - 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).
  - POST: Submits data to be processed (e.g., from an HTML form) to the identified resource, which may result in the creation of a new resource or the updates of existing resources or both.

How Does HTTP Work? (2)

- 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.
- DNS server: Resolves hostnames to IP addresses.
- Web server: Responds to HTTP requests.

How Does HTTP Work? (3)

- How do HTTP requests look like?
- Example: `http://www.google.com/search?q=ifis`

```
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-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Accept-Language: de,en;q=0.7
```

How Does HTTP Work? (4)

- Matching HTTP response of `www.google.com`:
  - Status code (200 means “resource found”)
  - Some information related to caching
  - MIME type of the resource
  - The resource itself

```
HTTP/1.1 200 OK
Date: Tue, 27 Jan 2009 10:03:57 GMT
Server: gws
Content-Type: text/html; charset=UTF-8
Content-Length: 498
Cache-Control: private, max-age=0
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Accept-Language: de,en;q=0.7
```

How Does HTTP Work? (5)

- Important types of HTTP requests are:
  - GET: Standard response for successful HTTP requests.
    - 200 (OK): Requests a representation of the specified resource.
    - 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.
    - 404 (Not Found): The resource has not been modified since last requested.
    - 410 (Gone): The requested resource could not be found (but may be available again in the future).
    - 500 (Internal Server Error): A server malfunction has occurred.

How Does HTTP Work? (6)

- Important types of HTTP status codes are:
  - 200 (OK):
  - 301 (Moved Permanently):
  - 302 (Found / Moved Temporarily):
  - 404 (Not Found):
  - 410 (Gone):
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 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: <h1>Main heading</h1> <h2>Sub Heading</h2>
  – Paragraphs: <p>Some text...</p>
  – Lists: <ul><li>First item</li><li>Second item</li></ul>
  – Links: <a href="http://www.google.com">Link to Google</a>

Currently, HTML is available in many different versions:
– 1995: HTML 2.0 (based on SGML)
– 1997: HTML 3.2
– 1997: HTML 4.0
– 1999: HTML 4.01
– 2000: "ISO HTML"
– 2000: XHTML 1.0 (based on XML)
– 2001: XHTML 1.1
– Current working draft: HTML 5 (close to final release)

Before 1989
– Hypertext and the Internet are separate, unconnected ideas

1989
– The English physicist Tim Berners-Lee is working at CERN, the European Organization for Nuclear Research, in Geneva
– He recognizes an important problem: Researchers from around the world needed to share data, with no common presentation software
– He wrote a proposal for "a large hypertext database with typed links", but it generated little interest, so he began implementing this system on a NeXT workstation
• 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 browser/editor: WorldWideWeb (runs only on NeXT)

• 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!”

• 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

A basic crawler (aka robot, bot, spider) consists of:
- A queue of URLs 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 indexer

The basic mode of operation:
1. Initialize the queue with URLs 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)
• 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
  – 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**

• 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

• 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!)

**Robot Exclusion Standard**

• 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: 
  – 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: 0800-0845

**MUST-Have Features (2)**

• 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: 0800-0845
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 crawlers 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: /

## 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: /

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

• 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

• Should-Have Features
### SHOULD-Have Features (2)

- **Freshness:**
  - The crawler should operate in continuous mode, i.e. it should obtain fresh copies of previously fetched pages
  - Crawl a page with a frequency that approximates the rate of change of that page
  - Be able to update a given set of pages on demand, e.g. if there is some current highly popular topic (“World Cup”)

- **Extensible:**
  - Be able to cope with new data formats, new protocols, ...
  - This amounts to having a modular architecture

### 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 (2)

- **Fingerprinting**
  - First, only use URIs in their normalized forms
    - This reduces the number of different URLs that must be handled
  - Then, for any normalized URI, compute its hash value (aka fingerprint) with respect to some hash function
    - A popular hash function is MD5, which can be computed quickly and yields a 128-bit fingerprint
    - Example of MD5: http://www.ifis.cs.tu-bs.de becomes 75924e8d184c52dd9bc5b368361093a8 (hexadecimal)
  - Now, build a B-tree (or hash table) of all fingerprints containing pointers to the original URIs

### The Duplicate URI Checker (3)

- **A B-tree:**
  [Diagram of a B-tree structure]
  - B-trees can be searched efficiently
  - Numerical comparisons can be done quickly
The whole process:

Is the given URI's fingerprint contained in the B-tree?

YES

NO

Is the given URI's fingerprint contained in the B-tree?

YES

NO

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.

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.

In principle, we could check for duplicate content in the same way as we did it for duplicate URIs.

But what about this page?

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

This problem is called near-duplicate detection.

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
Let \( \varphi \) be a random permutation on the set of all 64-bit integers, i.e., \( \varphi \) is a one-to-one function that maps any 64-bit integer to some 64-bit integer.

**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 rose”
  - “is a rose is”

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 rose”
  - “is a rose is”

**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.
  - 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.

**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 rose”
  - “is a rose is”

**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 (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 (9)**

- **Illustration:**
  - \( d = \text{“a rose is a rose is a rose”} \)
  - \( S(d) = \{ \text{“a rose is a”}, \text{“rose is a rose”}, \text{“is a rose is”} \} \)
  - \( H(d) = \{ 57892145, 110457815, 9235647 \} \)
  - \( \Pi(d) = \{ \text{57892145, 110457815, 9235647} \} \)
  - The simplest permutation is the identity mapping (every 64-bit number is mapped to itself)
  - A very clever indexing technique (to be discussed later)
  - 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.
    - 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 (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.
- 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 (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 \( S(d) \)

**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 (every 64-bit number is mapped to itself)
  - Another example of a permutation is \( \pi(x) = (x + 1) \mod 2^k \)
  - 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 \( \Pi(d) \)
- Furthermore, let \( \min(\Pi(d)) \) be the smallest number in \( \Pi(d) \)

**The Duplicate Content Checker (2)**

- **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 (every 64-bit number is mapped to itself)
  - Another example of a permutation is \( \pi(x) = (x + 1) \mod 2^k \)
  - 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 \( \Pi(d) \)
- Furthermore, let \( \min(\Pi(d)) \) be the smallest number in \( \Pi(d) \)

**The Duplicate Content Checker (2)**

- **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 (every 64-bit number is mapped to itself)
  - Another example of a permutation is \( \pi(x) = (x + 1) \mod 2^k \)
  - 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 \( \Pi(d) \)
- Furthermore, let \( \min(\Pi(d)) \) be the smallest number in \( \Pi(d) \)
We have to prove the following:

• What's the probability that both 
  \( S(d) \) and \( S(d') \) have their first "1" column at the same position?

Proof:

- First, represent the \( H(d) \) and \( H(d') \) as bit strings of length 2\(^{64} \), where \( H(d) \) is uniformly distributed over all permutations of columns:

\[
\Pr(H(d) = H(d')) = \Pr((min(\Pi(d)) = min(\Pi(d'))
\]

Hence, the overlapping of shingles can be expressed as the probability that the first "1" column of \( \Pi(d) \) is the same as the first "1" column of \( \Pi(d') \):

\[
\Pr(H(d) = H(d')) = \Pr(\text{first non-"0–0" column is a "1–1" column})
\]

Therefore:

- Since \( \Pi(d) \) is uniformly distributed over all permutations of columns,
  \( \Pr(\text{first non-"0–0" column is a "1–1" column}) = \frac{2^{64}}{2^{64}} = 1 \)

Hence, the overlapping of shingles is same as the probability that their first "1" column is in the same position.

\[
J(H(d), H(d')) = \Pr(\text{first non-"0–0" column is a "1–1" column})
\]

Note that \( J(H(d), H(d')) \) is usually very small, which is a random

\[
J(H(d), H(d')) = \Pr(\text{first non-"0–0" column is a "1–1" column})
\]

The Duplicate Content Checker (14)

The Duplicate Content Checker (12)

The Duplicate Content Checker (11)

The Duplicate Content Checker (10)

The Duplicate Content Checker (13)
Now back to our initial problem:

- Given:
  - A large collection of documents (and their pre-computed sketches)
  - A new document $d_{new}$
  - Near-duplicates of $d_{new}$ can be found by computing the sketch of $d_{new}$ and comparing to the sketches of all existing docs
  - This is much faster than computing shingles and their overlap

- But:
  - Finding near-duplicates is still quite expensive if we have to compare the sketch of every new document to all the sketches of the documents that already have been indexed
  - Linear complexity in the size of the index... But!

Possible 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 documents can be found by intersecting the sets of documents having at least $m$ numbers in common

Again, there is a trick:

- For each indexed document $d$ and each entry $(i, d)$ of its sketch $/g_2032(d)$, create a pair $((i, d), id(d))$, where $id(d)$ is $d$’s document id
- Finally, create a B-tree index that is sorted by the $id(d)$s
- Then, for each new document $d$, we can scan through its sketch and look up all other documents having at least one number in common—only these have to checked in detail...

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

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig

Information Retrieval and Web Search Engines — Wolf-Tilo Balke and Joachim Selke — Technische Universität Braunschweig
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(!)

Exploiting the Web graph for ranking

- HITS
- PageRank