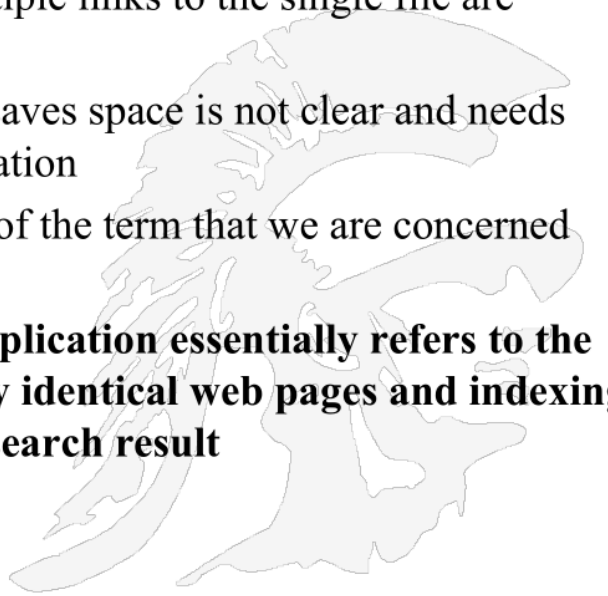
Deduplication

Definition [of deduplication]

- *De-Duplication* – the process of identifying and avoiding essentially identical web pages
- The term is often used in connection with *locker storage* where only a single copy of a file is stored and multiple links to the single file are managed
  - Whether this strategy effectively saves space is not clear and needs analysis for each particular application
  - However, this is ***not*** the meaning of the term that we are concerned about in this class
- **With respect to *web crawling*, de-duplication essentially refers to the identification of identical and nearly identical web pages and indexing only a single version to return as a search result**

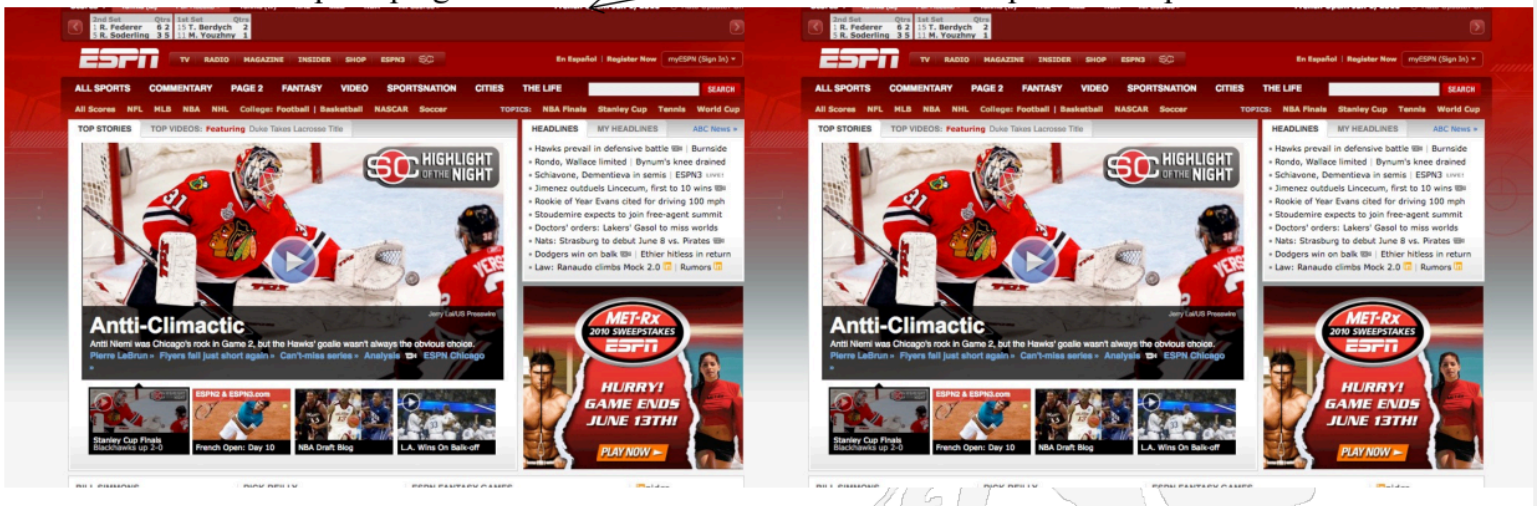


Duplicates

- One example is the same page, referenced by different URLs

http://espn.go.com

http://www.espn.com



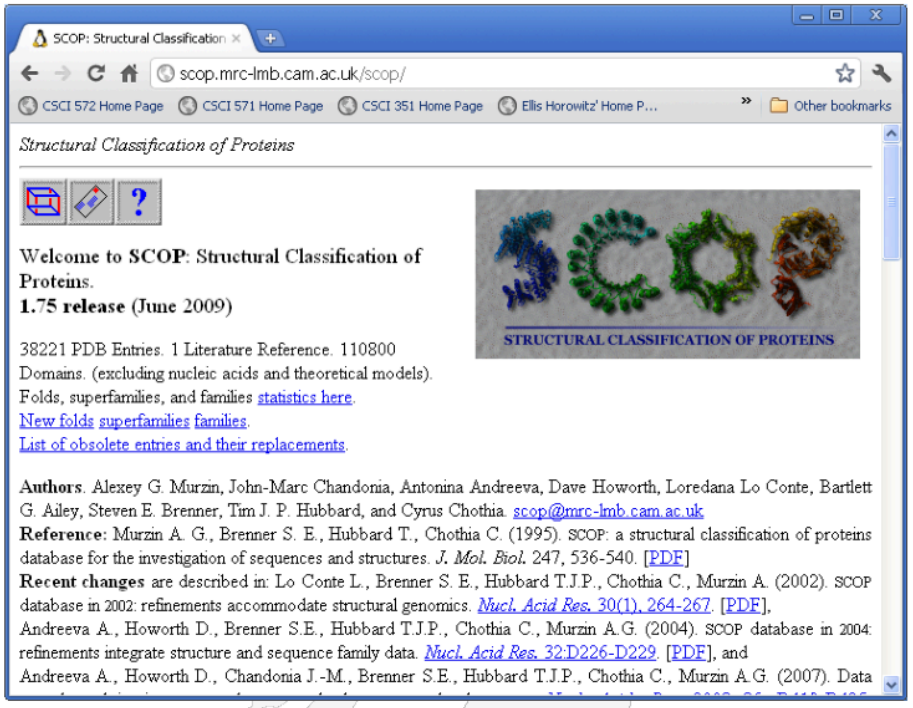
- How can two URLs differ yet still point to the same page?
  - the URL's host name can be distinct (virtual hosts) sharing the same document folder,
  - the URL's protocol can be distinct (http, https), but still deliver the same document
  - the URL's path and/or page name can be distinct

Totally distinct URLs – but same content

- At one time* all 3 URLs below pointed to the identical page
- Structural Classification of Proteins
  - http://scop.mrc-lmb.cam.ac.uk/scop
  - http://scop.berkeley.edu/
  - http://scop.protres.ru/

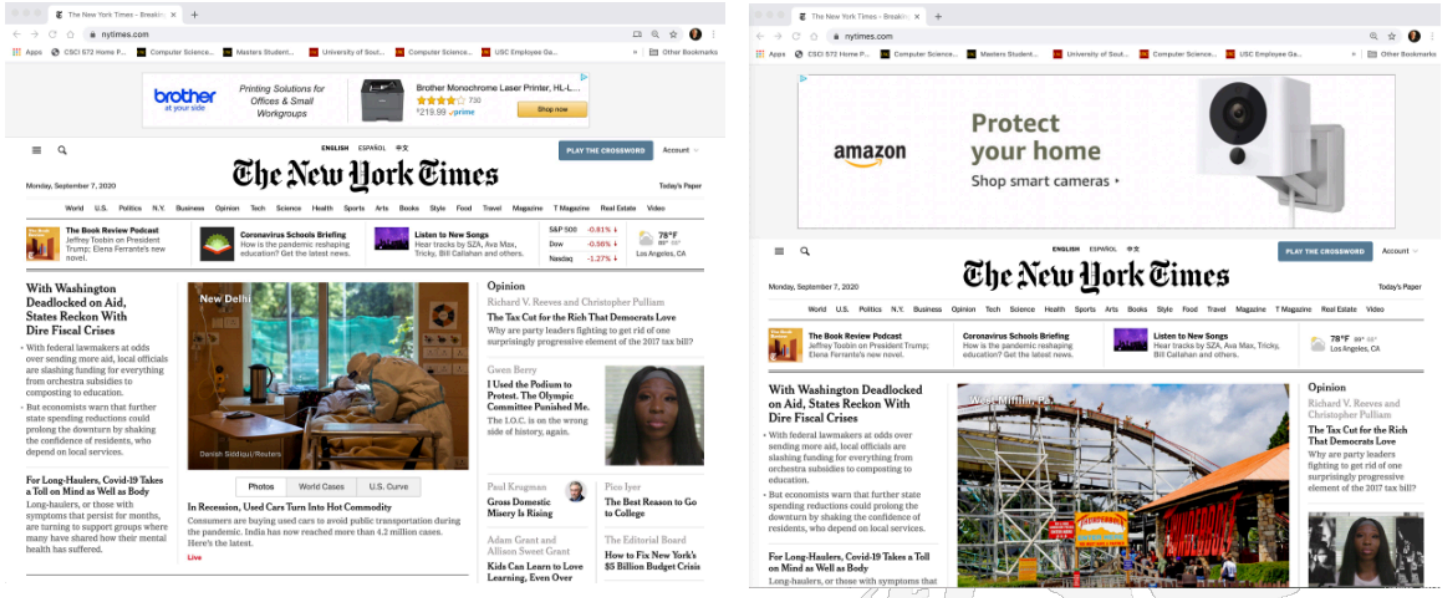
  - **The three URLs have distinct domain names, but all redirect to the same page**

*** At least they did when I took this snapshot, no longer**



'Near'-duplicates [almost identical]

Another example is two web pages whose content differs slightly



Two copies of www.nytimes.com snapshot within a few seconds of each other;
The pages are essentially identical except for the ads at the top and the photo in the middle

Spotting near-duplicates

- In examining a web page a search engine may want to ignore ads, navigation links and other elements that do not specifically relate to the contents of the web page
- One way to do this is to delve into the structure of a web page and focus on content blocks
- E.g. the Document Object Model for HTML displays a web page as a tree hierarchy
  - Document
  - Head
  - Body
- However this is time consuming

Duplicates: mirroring

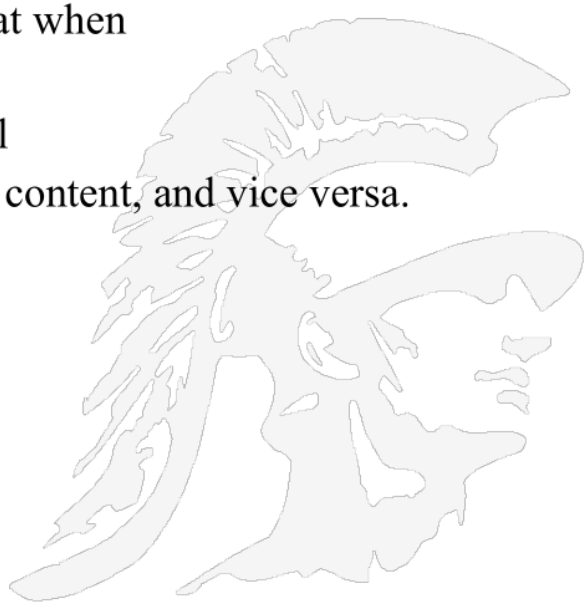
- Mirroring is the systematic replication of web pages across hosts.
  - Mirroring is the **single largest cause** of duplication on the web
- Host1/α and Host2/β are mirrors iff

  For all (or most) paths p such that when

  http://Host1/ α / p exists

  http://Host2/ β / p exists as well

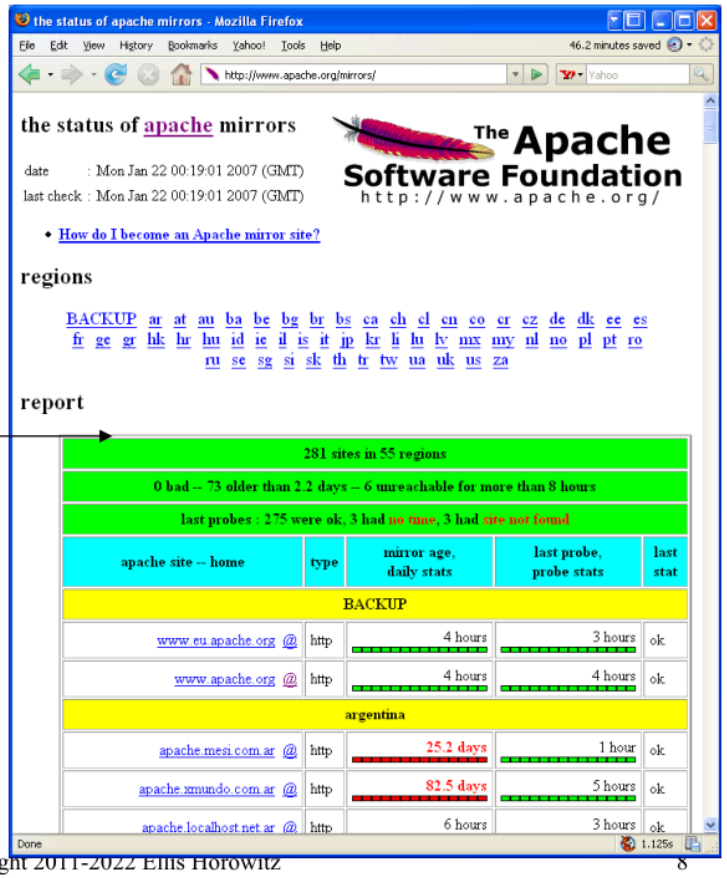
  with identical (or near identical) content, and vice versa.



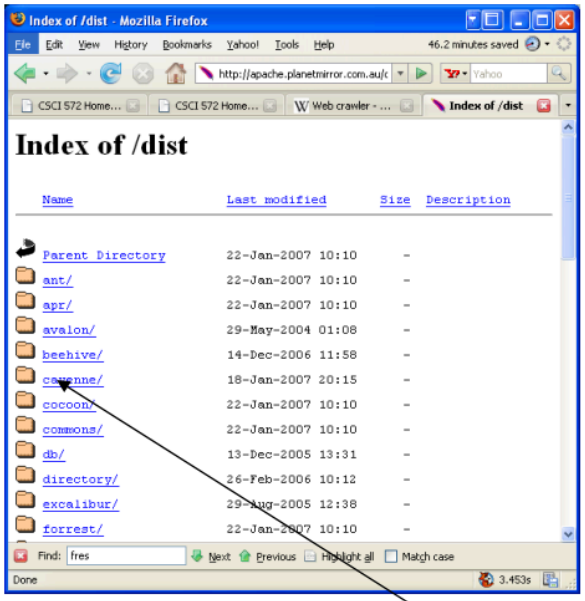
Apache – 'mirror' sites

List of countries

281 sites in 55 regions



Apache – two mirrors (for ex)



Site in Australia

Note identical directories

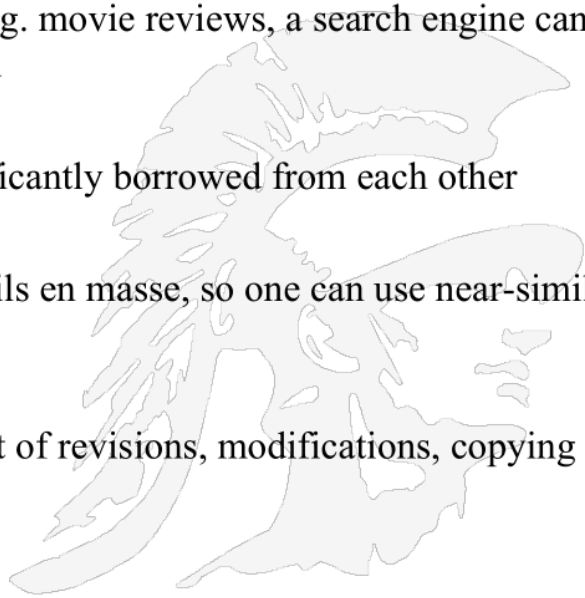
Site in Argentina

Why worry about exact duplicates?

- ***Smarter crawling***
  - Avoid returning many duplicate results to a query
  - Allow fetching from the fastest or freshest server
- ***Better connectivity analysis***
  - By combining in-links from the multiple mirror sites to get an accurate PageRank (measure of importance)
  - Avoid double counting out-links
- ***Add redundancy in result listings***
  - "If that fails you can try: <mirror>/samepath"
- ***Reduce Crawl Time***: Crawlers need not crawl pages that are identical or near identical
- ***Ideally***: given the web's scale and complexity, priority must be given to content that has **not** already been seen before or has recently changed
  - Saves resources (on the crawler end, as well as the remote host)
  - Increases crawler politeness
  - Reduces the analysis that a crawler will have to do later

Why worry about near-duplicates?

- **Clustering**
  - Given a news article some people might wish to see "related articles" describing the same event
- **Data extraction**
  - Given a collection of similar pages, e.g. movie reviews, a search engine can extract and categorize the information
- **Plagiarism**
  - Identify pairs that seem to have significantly borrowed from each other
- **Spam detection**
  - Spammers typically send similar emails en masse, so one can use near-similarity techniques to identify the spam
- **Duplicates within a domain**
  - To identify near-duplicates arising out of revisions, modifications, copying or merging of documents



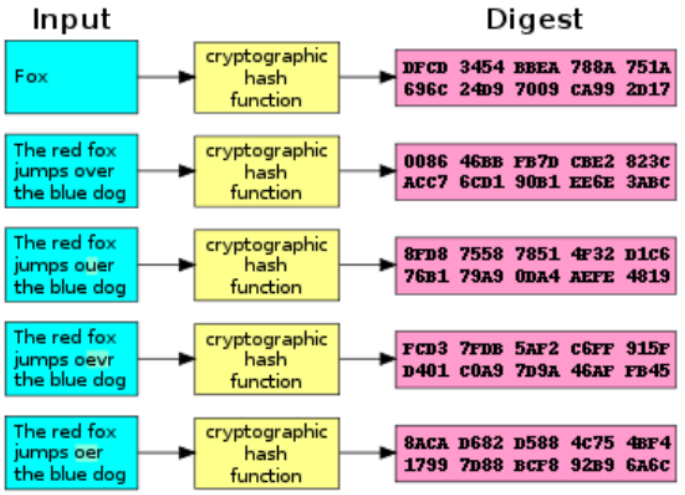
Solving the issue of duplicates/near-duplicates

1. ***Duplicate Problem*: Exact match;**
   - Solution: compute fingerprints using cryptographic hashing
   - Useful for URL matching and also works for detecting identical web pages
   - Hashes can be stored in sorted order for *log N* access
2. ***Near-Duplicate Problem*: Approximate match**
   - Solution: compute the syntactic similarity with an edit-distance measure, and
   - Use a similarity threshold to detect near-duplicates
     - e.g., Similarity > 80% => Documents are "near duplicates"

- The remaining slides are devoted to specific methods for duplicate and near duplicate detection

Cryptographic hash function: webpage -> number

- A **cryptographic hash function** is a hash function which takes an input (or 'message') and returns a fixed-size alphanumeric string, which is called the **hash value** (sometimes called a **message digest**, **digital fingerprint**, **digest** or a **checksum**).
- The cryptographic hash function has four main properties:
  1. It is extremely easy (i.e. fast) to calculate a hash for any given data.
  2. It is extremely computationally difficult to calculate an alphanumeric text that has a given hash.
  3. A small change to the text yields a totally different hash value.
  4. It is extremely unlikely that two slightly different messages will have the same hash.



A small change in a single word, "over" produces a major Change in the output; see https://en.wikipedia.org/wiki/Cryptographic_hash_function

Some popular cryptog hash fns

- The **MD5** (message-digest) hash function is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32 digit hexadecimal number.
  - Invented by Ron Rivest of MIT in 1991; replaced the earlier MD4
- The **SHA-1, SHA-2** hash functions are also quite popular (160 bit, 20 byte value)
  - SHA-1 was broken in 2005; using SHA-2 family of algorithms is now favored, see
  - https://en.wikipedia.org/wiki/SHA-2
- **SHA-3**, released in 2015; it produces digests of size 224, 256, 384 and 512 bits
- **RIPEMD-160** – a family of cryptographic hash functions and so far has not been broken; produces a 160 bit (20 byte) digest
- **E.g. See Chrome, Settings, Security and Privacy, Security, Manage certificates, certificates, Verisign**

https://bytes.usc.edu/~saty/tools/xem/run.html?x=MD5

Identifying identicals: 5 ways

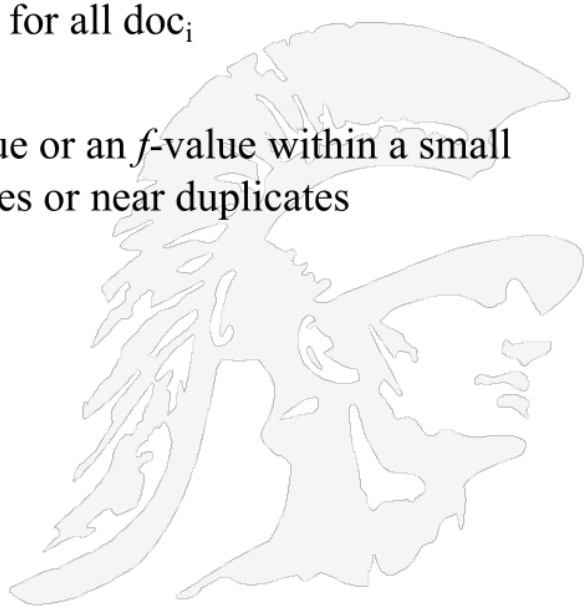
1. ***Compare character by character*** two documents to see if they are identical
   - very time consuming !!
2. ***Hash just the first few characters and compare*** only those documents that hash to the same bucket
   - But what about web pages where every page begins with <HTML> ??
3. ***Use a hash function*** that examines the entire document
   - But this requires lots of buckets
4. ***Better approach*** - pick some fixed random positions for all documents and make the hash function depend only on these;
   - This avoids the problem of a common prefix for all or most documents, yet we need not examine entire documents unless they fall into a bucket with another document
   - But we still need a lot of buckets
5. ***Even better approach***: Compute the cryptographic hash (SHA-2 or MD5) of each web page and maintain in sorted order, *O(log n)* to search

Identifying near-identicals: 2 ways

1. ***Produce fingerprints and test for similarity*** - Treat web documents as defined by a set of features, constituting an *n*-dimensional vector, and transform this vector into an *f-bit* fingerprint of a small size
   - Use Simhash or Hamming Distance to compute the fingerprint
     - SimHash is an algorithm for testing how similar two sets are
   - Compare fingerprints and look for a difference in at most k bits
   - E.g. see Manku et al., WWW 2007, *Detecting Near-Duplicates for Web Crawling,* http://www2007.org/papers/paper215.pdf
2. ***Instead of documents defined by n-vector of features, compute subsets of words (called shingles) and test for similarity of the sets***
   - Broder et al., WWW 1997, *Finding Near Duplicate Documents*

Identifying (near) duplicates – overall idea

1. Define a function *f* that captures the contents of each document in a number
   - E.g. hash function, signature, or a fingerprint
2. Create the pair $<f(doc_i), ID\ of\ doc_i>$ for all doc_i
3. Sort the pairs
4. Documents that have the same *f*-value or an *f*-value within a small threshold are believed to be duplicates or near duplicates

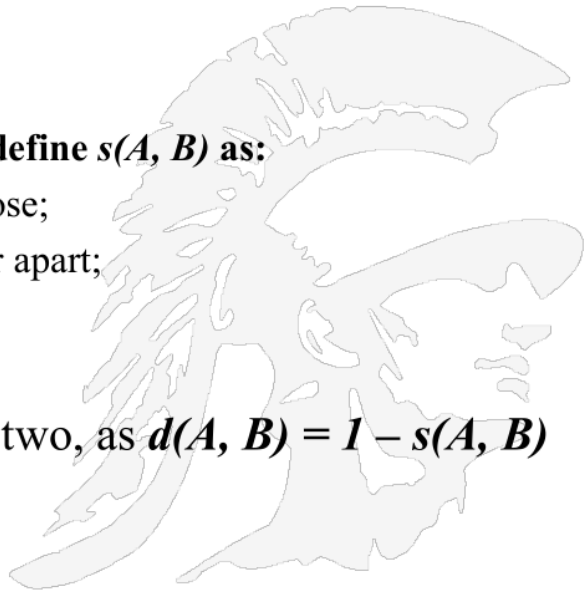


Distance measures (to compute similarity)

- **To compute similarity, we need a distance measure**
- **A distance measure must satisfy 4 properties**
  1. No negative distances
  2. *D(x,y) = 0 iff x=y*
  3. *D(x,y) = D(y,x)* symmetric
  4. *D(x,y) <= D(x,z) + D(z,y)* triangle inequality
- **There are several distance measures that can play a role in locating duplicate and near-duplicate documents**
  - *Euclidean distance* – D([$x_1 \ldots x_n$], [$y_1, \ldots, y_n$]) = sqrt(Sum(x_i-y_i)^2) i=1…n
  - *Jaccard distance* – D(x,y) = 1 – SIM(x,y) or 1 minus the ratio of the sizes of the intersection and union of sets x and y
  - *Cosine distance* – the cosine distance between two points (two *n* element vectors) is the angle that the vectors to those points make; in the range 0 to 180 degrees
  - *Edit distance* – the distance between two strings is the smallest number of insertions and deletions of single characters that will convert one string into the other
  - *Hamming distance* – between two vectors is the number of components in which they differ (usually used on Boolean vectors)

Set distance, set similarity - related measures

- **A set is an unordered collection of objects, e.g. {a, b, c}**
- **Focusing on the notion of *distance* of two sets we define a distance *d(A, B)* as**
  - *small,* if objects in A and B are close;
  - *large*, if objects in A and B are far apart;
  - *0,* if they are the same, and finally
  - *d(A, B)* is in the range [0, infinity]
- **Focusing on the notion of *similarity* we define *s(A, B)* as:**
  - *large,* if the objects in A and B are close;
  - *small,* if the objects in A and B are far apart;
  - *1,* if they are the same, and finally
  - *s(A, B)* is in the range [0, 1]
- Often we can convert between the two, as ***d(A, B) = 1 – s(A, B)***



'Jaccard similarity/index'

- **Consider *A = {0, 1, 2, 5, 6}* and *B = {0, 2, 3, 5, 7, 9}***
- ***JS(A, B) = size(A intersection B) / size(A union B)***
- ***= size({0, 2, 5}) / size({0, 1, 2, 3, 5, 6, 7, 9})***
- ***= 3 / 8 = 0.375***
- **Suppose we divide our items into four clusters, e.g**
  - **$C_1 = \{0, 1, 2\}$**
  - **$C_2 = \{3, 4\}$**
  - **$C_3 = \{5, 6\}$**
  - **$C_4 = \{7, 8, 9\}$**

perhaps
C_1 represents action movies, C_2 comedies,
C_3 documentaries, C_4 horror movies

- **If $A_{clu} = \{C_1, C_3\}$ and $B_{clu} = \{C_1, C_2, C_3, C_4\}$, then**
- **$JS_{clu}(A, B) = JS(A_{clu}, B_{clu})$ =**
- **size (({C1, C3} intersect {C1, C2, C3, C4}) / ({C1, C3} union {C1, C2, C3, C4}))**
- **= 5 / 10 = 0.5**
- **If we are going to use Jaccard similarity to determine when two web pages are near duplicates; we need to say what are the elements of the sets we are comparing**

Jaccard similarity, from sets of shingles

- **Definition of Shingle**:
  - a contiguous subsequence of words in a document is called a *shingle;* The 4-shingling of the phrase below produces a bag of 5 items:

    "a rose is a rose is a rose" => a set S(D,w) is defined as

    { (a_rose_is_a), (rose_is_a_rose), (is_a_rose_is), (a_rose_is_a), (rose_is_a_rose)}
  - *S(D,w)* is the set of shingles of a document *D* of width *w*
- **Similarity Measures**
  - *Jaccard(A,B)* (also known as Resemblance) is defined as

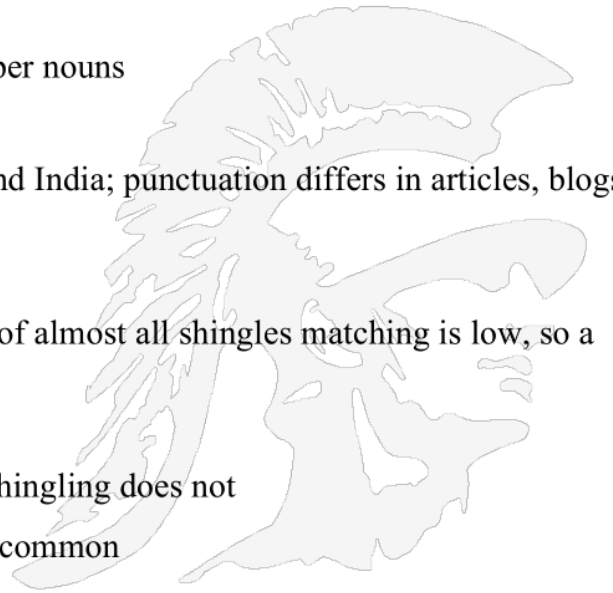
    size of (S(A,w) intersect S(B,w)) / size of (S(A,w) union S(B,w))
  - *Containment(A,B)* is defined as

    size of (S(A,w) intersect S(B,w)) / size of (S(A,w))
  - 0 <= Resemblance <= 1
  - 0 <= Containment <= 1

- See *On the resemblance and containment of documents*, Conf. on Compression and Complexity, DEC Research Center, 1997

Shingling – choices

- **White space?**
  - Should we include spaces and returns? Sometimes it makes sense, e.g.
    "plane has touch down" versus "threw a touchdown"
    (the space between "touch" and "down" is significant)
- **Capitalization?**
  - Sam versus sam. Can help to distinguish proper nouns
- **Punctuation?**
  - English is punctuated differently in the US and India; punctuation differs in articles, blogs, and tweets
- **How large should *k* be?**
  - General rule: high enough so the probability of almost all shingles matching is low, so a collision is meaningful;
- **Count replicas?**
  - Typically bag of words counts replicas, but shingling does not
- **Stop words?** Typically omitted as they are so common



Shingles -> numbers

- **Original text**
  - "Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species"
- **All 3-shingles (there are 16 of them)**
  - (Tropical fish include), (fish include fish), (include fish found), (fish found in), (found in tropical), (in tropical environments), (tropical environments around), (environments around the), (around the world), (the world including), (world including both), (including both freshwater), (both freshwater and), (freshwater and salt), (and salt water), (salt water species)
- **Hash values for the 3-shingles (sets of shingles are large, so we hash them to make them more manageable, and we select a subset)**
  - 938, 664, 463, 822, 492, 798, 78, 969, 143, 236, 913, 908, 694, 553, 870, 779
- **Select only those hash values that are divisible by some number, e.g. here are selected hash values using *0 mod 4***
  - 664, 492, 236, 908; ***these are considered the fingerprints***
- **Near duplicates are found by comparing fingerprints and finding pairs with a high overlap**

Jaccard similarity, and shingling

- Recall the Jaccard similarity of sets A and B, *J(A,B)*, is defined as

  | *A intersect B* | / |*A union B* |
- The Jaccard distance of sets A and B, measuring *dis*similarity is defined as

  1 - J(A,B), or equivalently {(A union B) - (A intersect B)}/(A union B)
- We can test if two pages are near duplicate by
  1. First compute the *k*-shingles of the two pages
  2. Map the k-shingles into numbers (e.g. by hashing)
  3. Select a subset of the shingles to act as the fingerprints
  4. Compute the Jaccard similarity of the *k*-shingle fingerprints;
- **A high Jaccard similarity (e.g. greater than 0.9), implies the pages are near duplicate; or**
- ***if (J (fingerprint(A), fingerprint(B))) > k*, then the pages are similar;**

FYI if you are interested: in p.475 of our text is a probabilistic approach to this that reduces the # of shingle comparisons we need to make.

SimHash

- There is another way to determine if two web pages are near duplicates
- The method is called SimHash
- It was developed by Moses Charikar and is described in his paper *Similarity Estimation Techniques from Rounding Algorithms,* STOC May 2002
  - https://www.cs.princeton.edu/courses/archive/spring04/cos598B/bib/CharikarEstim.pdf
- The basic idea is the same as before
  - obtain an *f*-bit fingerprint for each document
  - A pair of documents are near duplicate if and only if fingerprints are at most *k*-bits apart
  - But in this case instead of using permutations and probability we use SimHash
- Documents D_1 and D_2 are near duplicates *iff*

  $Hamming\text{-}Distance(Simhash(D_1), Simhash(D_2)) <= K$
- Typically $f = 64$ and $k = 3$

SimHash (aka Charikar Similarity) is essentially a dimension reduction technique - it maps a set of weighted features (contents of a document) to a low dimensional fingerprint, eg. a 64-bit word.

And, **documents that are nearly identical have nearly similar fingerprints that differ only in a small # of bits.** In other words, similar inputs lead to similar outputs (hash values), hence 'Sim'Hash; other hashing techniques, eg. MD5, do not have this property (in other words, even a tiny change in the input leads to a huge change in the output). This similarity property is what makes SimHash, an excellent tool for similarity detection of documents.

Here is the SimHash paper.

SimHash: LSH [Locality-Sensitive Hashing]

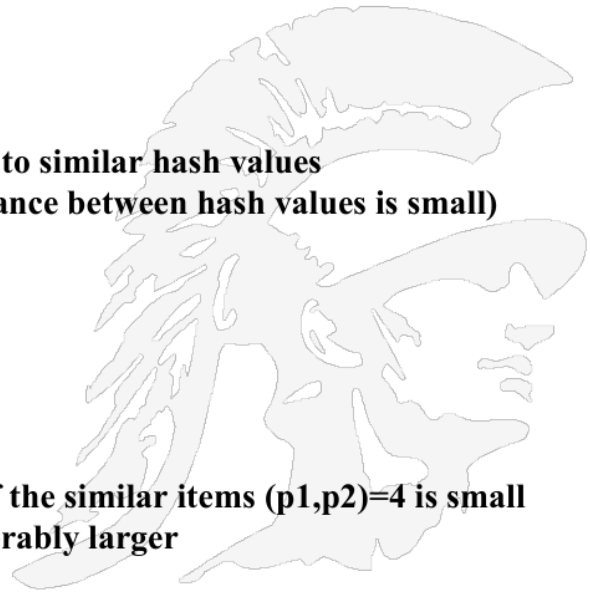
- A hash function usually hashes different values to totally different hash values; here is an example

  p1 = 'the cat sat on the mat'
  p2 = 'the cat sat on a mat'
  p3 = 'we all scream for ice cream'
  p1.hash => 415542861
  p2.hash => 668720516
  p3.hash => 767429688

- Simhash is one where similar items are hashed to similar hash values
  (by similar we mean the bitwise Hamming distance between hash values is small)

  p1.simhash => 851459198
  00110010110000000011110001111110
  p2.simhash => 847263864
  00110010100000000011100001111000
  p3.simhash => 984968088
  00111010101101010110101110011000

- in this case we can see the hamming distance of the similar items (p1,p2)=4 is small whereas (p1,p3)=16 and (p2,p3)=12 are considerably larger



SimHash algorithm

- **The simhash of a phrase is calculated as follows:**

1. **pick a hashsize, lets say 32 bits**
2. **let V = [0] * 32 # (ie a vector of 32 zeros)**
3. **break the input phrase up into shingles, e.g.**
   'the cat sat on a mat'.shingles(2) =>
   #<Set: {"th", "he", "e ", " c", "ca", "at", "t ", " s", "sa", "at", "t ", " o", "on", "n ", " a", "a ", " m", "ma", "at"}>
4. **hash each feature using a normal 32-bit hash algorithm (MD5 or SHA)**
   "th".hash = -502157718 "he".hash = -369049682 ...
5. **for each hash**
   if bit_i of hash is set then add 1 to V[i]
   if bit_i of hash is not set then subtract 1 from V[i]
6. **simhash bit_i is 1 if V[i] > 0 and 0 otherwise**

- **Simhash is useful because if the Simhash bitwise Hamming distance of two phrases is low then their Jaccard coefficient is high**

SimHash algorithm [cont'd]

- **In the case that two numbers have a low bitwise Hamming distance and the difference in their bits are in the lower order bits then it turns out that they will end up close to each other if the list is sorted.**
- **consider the eight numbers and their bit representations**
- **1 37586 1001001011010010**
- **2 50086 1100001110100110 7 <--(this column lists hamming**
- **3 2648 0000101001011000 11 distance to previous entry)**
- **4 934 0000001110100110 9**
- **5 40957 1001111111111101 9**
- **6 2650 0000101001011010 9**
- **7 64475 1111101111011011 7**
- **8 40955 1001111111111011 4**

if we sort them

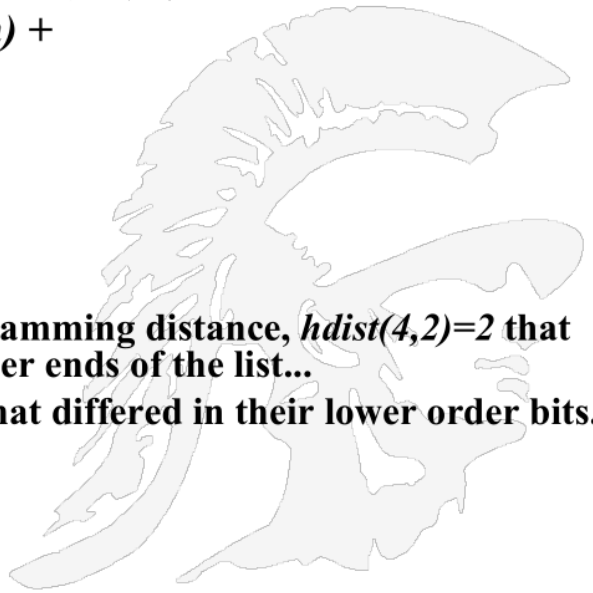
4 934 0000001110100110
3 2648 0000101001011000 9
6 2650 0000101001011010 1
1 37586 1001001011010010 5
8 40955 1001111111111011 6
5 40957 1001111111111101 2
2 50086 1100001110100110 9
7 64475 1111101111011011 9

notice that two pairs with very smallest hamming distance hdist(3,6)=1 and hdist(8,5)=2 have ended up adjacent to each other.

SimHash algorithm [cont'd]

- **Rather than check every combo we could just check the adjacent pairs of the list, each is a good candidate.**
- **This reduces the runtime from *n*(n-1)/2* coefficient calculations, $O(n^2)$ to**
  - ***n* fingerprints calculations *O(n)* +**
  - **a sort *O(n log n)* +**
  - ***n* coefficient calculations *O(n),***
- **which is *O(n log n)* overall;**

- **A problem:**
  - **there is another pair with a low Hamming distance, *hdist(4,2)=2* that have ended up totally apart at other ends of the list...**
  - **sorting only picked up the pairs that differed in their lower order bits.**



SimHash algorithm [cont'd]

- **To get around this consider another convenient property of bitwise Hamming distance, *a permutation of the bits of two numbers preserves Hamming distance***
- **If we permute by 'rotating' the bits, i.e. bit shift left and replace lowest order bit with the 'lost' highest order bit we get 'new' fingerprints that have the same Hamming distances**

'rotate' bits left twice

4 3736 0000111010011000
3 10592 0010100101100000 9
6 10600 0010100101101000 1
1 19274 0100101101001010 5
8 32750 0111111111101110 6
5 32758 0111111111110110 2
2 3739 0000111010011011 9
7 61295 1110111101101111 9

if we sort again by fingerprint

4 3736 0000111010011000
2 3739 0000111010011011 2
3 10592 0010100101100000 11
6 10600 0010100101101000 1
1 19274 0100101101001010 5
8 32750 0111111111101110 6
5 32758 0111111111110110 2
7 61295 1110111101101111 6

this time the (2,4) pair ended up adjacent
we also identified the (3,6) and (5,8) pairs as candidates again

So we can **'rotate, sort, check adjacent'** 'B' times (eg. 64 times; depending on how many bits we have), to discover (almost) all the near-duplicates.

In other words:

- comparing SimHash values (ie computing 'Charikar Similarity' values) is a great way to identify near-duplicates
- for 'n' documents, comparing them all pairwise would take a long time [O(n^2)]
- so as a shortcut, we can sort their decimal representations and only compare adjacents - this will identify similarities based on low-end bits; but this will miss similarities based on the higher-end bits; as an aside, we can look for one more possible low-bits near-duplicate by comparing the top-most and bottom-most values too, like in Gray Code
- so to fix the problem of missing finding high order bit similarities, we can rotate (spin) all the docs' bits identically to the right (so that the high order bits become a 'bit' (lol) lower) to produce 'new' hashes, sort *those*, compare for near-duplicates
- we can progressively spin right by 1 bit, 2 bits, 3 bits... to discover more and more similarities [we will rediscover existing similarities but ignore those]
- note that we can spin left as well
- doing the above is STILL faster than O(n^2) :)

Here is a nice page on SimHash, and this is a Google paper that discusses using SimHash at scale.