Brief introduction to web-searching

with an overview of

information retrieval techniques

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. All rights reserved.

What is a document? Definition

- A **document** is a single unit of information in digital form.
- Documents are usually represented through a set of index terms or **keywords**.
- Keywords can be automatically extracted from the document or be user-specified (eg. an expert decides them). The keywords form the **logical view** of a given document.
- **Full-text** logical view is one in which all the words of a document become keywords.
- Because of space problems, an **index-term** logical view is more preferable where keywords are further refined to a smaller set of index-terms. This can be accomplished by the use of **text operations** that
  - eliminate **stopwords** such as articles and connectives,
  - use **stemming** to reduce distinct words to their common grammatical root,
  - identify **noun groups**, and
  - identify **similar (in meaning) words** called **synonyms**.
- To accommodate such a view in a way that is viable, **compression** might also be used.

(c) Copyright A. Gerbessiotis. All rights reserved.

Measuring the Web according to Google @ 2000

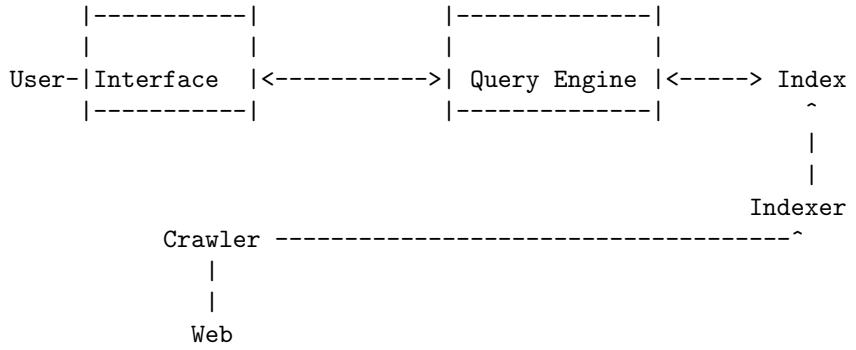
```
Google's Crawler @ Stanford [circa 2000]
  Active crawlers       :  3-4 typically
  Open Connections      : 300
  Web-page read         : 100 concurrently
  Data                  : 600Kbytes/second

Google @ Stanford size statistics (in millions)
  # of Web pages        : 24
  #  of URLs            : 76
  #  of Email addresses : 1.7
  # of 404messgs        : 1.6

Google Index Data
  Fetched Information   : 147Gbytes
  Compressed            :  53Gbytes (3 to 1)
  Inverted Index Full   :  37Gbytes
  Lexicon               :  .3Gbytes
  DocIndex              : 9.7Gbytes
  links                 : 3.9Gbytes

   All data from Brin & Page [ http://www-db.stanford.edu/~backrub/google.html ]
```

(c) Copyright A. Gerbessiotis. All rights reserved.



A Centralized Architecture (eg. Altavista)

Crawler: Collects information by following hyperlinks.
Indexer : Creates data structure for fast searching of documents based on the indexed information.
Query Engine: Interacts with query interface (and thus the user) and also the index to find hits, and rank the results and also provide summary information.
Interface : User entry point to the system.

(c) Copyright A. Gerbessiotis. All rights reserved.

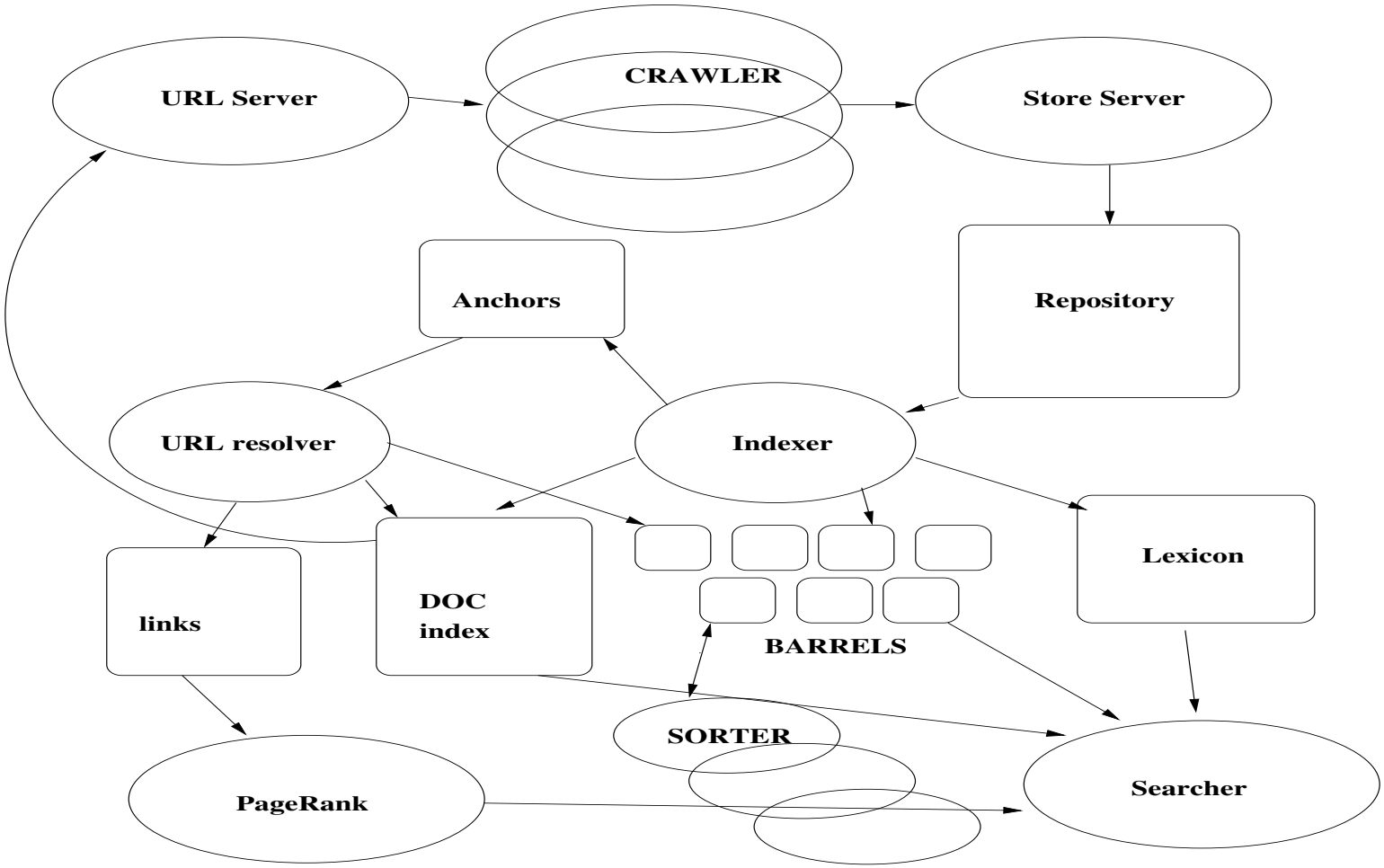
An example of a Retrieval Process: Google

1. Parse the query.
2. Convert words into wordIDs.
3. Seek to the start of the doclist in the short barrel for every word.
4. Scan through the doclists until there is a document that matches all the search terms.
5. Compute the rank of that document for the query.
6. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
7. If we are not at the end of any doclist go to step 4.
8. Sort the documents that have matched by rank and return the top k.

Figure (Google Query Evaluation) from http://www-db.stanford.edu/~backrub/google.html

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Google System Architecture



(c) Copyright A. Gerbessiotis. All rights reserved.

Google System Architecture explained

URL server. Provides a list of URLs to be sent to and retrieved by the crawler.

Crawler. A distributed crawler is used with 3-4 instances running at any time (in 1998-2000).

StoreServer. Compresses Web-pages (using zlib) and then stores them into the

Repository, that holds all retrieved pages. The indexer can use the Repository and the error log of the Crawler to restore the index.

Indexer. Parses documents and generates Google's index.

Anchors. Contains a list of (possibly relative) URLs retrieved by the Indexer.

URLresolver. Converts Anchors-related data into absolute URLs and uses this information to generate the

Links, data that show how URLs reference each other (i.e. depict the link structure of the Web).

Lexicon. Contains a list of unique words found (14million, 256MBytes of data in 2000). Resides in main memory (and the size of the main memory determines its size).

DocIndex. Contains information about documents. Every document has a docID which is a hash of its contents, a pointer to its location to the Repository along with other useful information (eg. URL and title).

Barrels. Contain all hitlists generated by the indexer from the Repository. Google originally used 64 Barrels, one barrel for a group of consecutive words. It contains information about hits i.e. is an inverted index of all words in all documents (every word records also positional, font and TITLE/ANCHOR relative indexing information).

Sorter. It's the program that sorts the Barrels based on word (actually wordID) to generate the hitlists.

PageRank. It's the ranking method used by Google to rank documents based on "popularity". Such information is based solely on the graph structure available in Links.

Searcher. It's the search interface for Google. It parses queries, converts them into a form google understands (e.g. converts words into wordIDs using the Lexicon), accesses the Barrels to determine the documents. The ranking of the output documents is a combination of PageRank and an evaluation of the results of the hits according to the query.

(c) Copyright A. Gerbessiotis. All rights reserved.

Consider searching a small collection of HTML documents that describe Computer Science books. Each such document minimally includes the authors and the title of the book it describes. Let us for the sake of this example we limit our exposure on few (approximately 10) books on say `data structures` and `algorithms`. For this collection the index-terms will be limited to this domain.

Through the document processing techniques of the previous section we decide that the index-terms are relevant words such as `algorithms, algorithm, data structures, design, analysis, computer, java, c++`. One might observe that stemming was not used and thus `algorithm` and `algorithms` are considered two different index-terms. One might also observe that a noun-group was identified and thus `data structures` became an index-term (not index-terms). For easiness we shall refer to each document in the remainder by the author(s) of the book that it describes rather than a generic and nameless `docID`.

If we are interested in searching book titles, one way to organize the information collected through document processing (subject 5) is to create for each index-term an `incidence vector`. An `incidence vector` for a term is a binary vector of indicator hits for a sequence of documents. In other words for a term t its incidence vector C_t is a vector that is D elements long (where D is the number of documents in the collection) and entry $C_t(i)$ is 1 if the i-th document of the collection contains term t and 0, if the i-th document of the collection does not contain t.

An `indicator` variable is 1 if a condition is true and 0 otherwise. Thus $C_t(i)$ is the indicator variable for term t appearing in document i.

We can group the incidence vectors of all index-terms in the form of a matrix i.e. a collection of rows and columns of values (0s and 1s in our case). The rows in the matrix are the books i.e. `documents` of the collection, and the columns are terms and their incidence vectors. If we denote a matrix by C and reference it as $C(i, j)$ where index i references a row and index j references a column, then $C(i, j)$ is 1 if the j-th term appears in the i-th document and 0 otherwise. Thus the j-th column of C is the incidence vector for the j-th term.

An incidence matrix is depicted in Figure . The rows in the figure are the documents and describe the index-terms appearing in the documents and the columns the incidence vectors. The term `algorithms` has an incidence vector of `0101111111` since all but the first and third books contain it. This incidence vector is the first column of the C since `algorithm` is its first index-term.

(c) Copyright A. Gerbessiotis. All rights reserved.

	Algorithms	Algorithm	Data Structures	Java	C++	Computer	Design	Analysis
Knuth	0	0	0	0	0	0	1	0
Cormen et al	1	0	0	0	0	0	0	0
Tamassia et al	0	1	0	0	0	0	1	0
Aho et al	1	0	0	0	0	1	1	1
Baase et al	1	0	0	0	0	1	0	0
Lewis et al	1	0	1	0	0	0	0	0
Aho et al (2)	1	0	1	0	0	0	0	0
Goodrich et al	1	0	1	1	0	0	0	0
Goodrich et al(2)	1	0	1	0	1	0	0	0
Levitrin	1	0	0	0	0	0	1	1

Given an incidence matrix C for a corpus (document collection) query processing involves manipulating the columns of C i.e. its incidence vectors using boolean operators (i.e. `AND, OR, NOT`).

In order for example to respond to a query of the form

`algorithms` AND `data structures`

we form the `bitwise AND` operation of the incidence vectors of the two terms `algorithms` and `data structures`.

The bitwise AND of two n-element vectors x, y of 0s and 1s is an n-element vector z of 0s and 1s denoted by $z = x\&y$ such that the i-th element of z is 1 if and only if the i-th elements of x and y are both 1, and 0 otherwise. (For a bitwise OR denoted by $z = x|y$, the i-th element of z is 1 if one or both of the i-th elements of x and y are one.)

algorithms. The incidence vector of `algorithms` is `x = 0101 1111 11`.

data-structures. The incidence vector of `data structures` is `y = 0000 0111 10`.

Then $z = x\&y =$ `0000 0111 10`. (In this example we group bits into groups of four with a space in between to make the vectors more readable.)

This means that the sixth, seventh, eighth, ninth books contain both terms in their titles!

(c) Copyright A. Gerbessiotis. All rights reserved.

Requirements and problems

Consider the case of Google in 1997 rather than the case of Bing or Google nowadays. In 1997 Google used to index as few as 25,000,000 books and thus $D = 25000000$, and the dictionary that contained the index-terms contained close to $n = 14,000,000$ words.

- **Corpus size.** $150GB$.
- **Number of documents.** $D = 25,000,0000$.
- **Number of index-terms.** $n = 14,000,000$.
- **Average document size.** $150GB/25000000 = 6KB$.
- **Average word length.** 8.
- **Average number of words per document.** $6KB/8 \approx 1000$.

Collecting some data from subject 2, we realize that every document contains on the average no more than 1000 words yet an incidence vector is $25,000,000$ element long, and the incidence matrix thus contains $14,000,000$ such incidence vectors.

- **Incidence matrix C is a** $D \times n$ matrix where $D \times n = 25,000,000 \times 14,000,000$ matrix.
- **On the average, an incidence row** has 1000 non-zero values (1s) and $13,999,000$ zero values.

The total amount of space used to store the incidence matrix (assuming one bit for a single zero or one) is $25 \cdot 10^6 \times 14 \cdot 10^6/8 = 4.3 \cdot 10^{13}$ bytes, or roughly $43TB$ (i.e. **43 Terabytes**).

Yet, Google in 1997 indexed those same documents in around **50 Gigabytes**, an amount of space that is roughly one-thousandth (**50 Gigabytes** vs **43 Terabytes**) of the incidence matrix space.

Sparse form of incidence matrix $\rightarrow$ **Inverted Index**!

(c) Copyright A. Gerbessiotis. All rights reserved.

Thus an incidence matrix is a very poor solution for storing index information. This is because it stores not only the 1s (useful information) but also the 0s. The zeroes tell us nothing: they indicate that an index-term does not appear in a document. It makes thus more sense to store only the 1s and forget the 0s. That is, for every document we store its index-terms (i.e. the 1s) contained in it and ignore the 0s. This is an `orthogonal/inverse` solution!

Orthogonally (or inversely), for every index term, store all the documents that contain that index term in a data structure aptly named `inverted index`!

An inverted index would thus store only $25,000,000 \times 1000$ document references. A document reference (i.e. `docID`) needs approximately 4 bytes for a total of $100GB$. With some compression this space can easily be halved into 50GB, very close to Google's index size in 1997. (Those of you familiar with the concepts of an `adjacency matrix` and an `adjacency list` representation of a graph, might identify their similarities to incidence matrices and inverted index.)

An **inverted index** is a word-oriented mechanism for indexing a text collection and thus storing index-term or document-incidence information in order to speedup the search task.

An inverted index consists of two components.

- A **Vocabulary**, the set of all index-terms in the dictionary, and
- **Occurrence lists** or **inverted lists** that contain the information necessary for each index-term of the vocabulary to be identified.

An **index of a book**, contains a list of important words (keywords), and for each such word the page in which it appears is listed. If a word appears multiple times in a page, that page is listed once. If consecutive pages include the word, a compressed reference is used that includes the first page and the last page of the sequence separated with a dash $-$. The list of words in the index is a small subset of the words of the book, and potentially a smaller fraction of the dictionary of all English words (i.e. those of the Oxford English Dictionary, Second Edition).

An **index of a corpus** might contain for each word, the document in which it appears, and/or positional information within that document (i.e. page numbers). The level of granularity might depend on the use of the index.

In several books or collections of books, a precursor to an inverted index and also to those two instances of an index is what are known as **concordances**, a primitive form of an occurrence list.

What information to record?

Dictionary vs Vocabulary vs Lexicon. All three mean the same thing, a collection of unique words, but in the context of this discussion we are going to use them in a specific way. So in the context of subject 6, a **dictionary** contains all the unique words in the corpus as identified by the tokenizer. For example a dictionary might contain `auto, car, vehicle, automobile`. The vocabulary contains all the unique index-terms that are indexed and thus included in the inverted index. In the dictionary all four words that identify a car might all point to the same index-term of the vocabulary, say `car`. For example for a query that includes the word `auto`, a lookup to the dictionary can resolve the correct spelling of this word (as opposed to say, `awto`) and also identify the correct index-term associated with the word i.e. `car`. Then `car` will replace `auto` in the query before it is being further processed. It is often that dictionary and vocabulary are one and the same if only because of the overhead of maintaining two different data structures for them. The **lexicon** is an efficient implementation of the dictionary/vocabulary by using a hash table more often than not. It is possible that in this context not all of the words of a vocabulary (or dictionary) are stored into the hash table because of memory requirements (not enough available memory to store all the words in main memory). Thus in this case some rare words (of the vocabulary or dictionary) might be stored outside of the lexicon (i.e. outside of main memory).

Inverted list/Occurrence list for a given index-term holds the relevant information for the index-term. Each entry in the list is called a **posting** and the portion of a posting that refers to a specific document or location is called a **pointer**. The posting minus its pointer might record additional information about the index-term. In the case of Google (1997) it records, things such as relative font-size and capitalization.

Hitlist refers to the occurrence list of particular index-term in a particular document. Thus the hitlists of an index-term over all documents of the collection form its occurrence list. For example, an index-term that is contained in 10 documents will consist of 10 hitlists. Each hitlist is going to be (or can be) of variable length depending on the times the index-term appears in the associated document.

Doclist refers to a simplified form of an occurrence list (inverted list) in which only document references appear (i.e. posting and pointer coincide and only include a document reference). (For example, Fig. 5.3 of the textbook on page 132 gives a doclist.)

Note that in Google (1997) an occurrence list is a collection of doclists with associated hitlists. Thus variations of the given explanations might be (and are) prevalent.

(c) Copyright A. Gerbessiotis. All rights reserved.

A. Information recorded in a vocabulary. An implementation of a vocabulary can be called a lexicon; how a lexicon can be implemented was the subject of Subject 4 following the example of Google (1997). However in the discussion to follow we are not interested in implementation (i.e. data-structure) details, but we describe only relevant information that one would be interested in maintaining in a vocabulary (or its implementation, a lexicon).

A1. WordID. One entry per index-term. Rarely the word itself is stored in the vocabulary. A hash in the form of a word identifier called `wordID` is more useful to maintain and compact to store. A pointer to an area in memory where the string of the word itself is stored might also be available.

A2. Ndocs. The number of distinct documents (e.g. URLs or `docID`s) containing the word `wordID` can be recorded in this field.

A3. Nhits. The number of distinct occurrences of `wordID` in all `Ndocs` documents or separately per document can be recorded in this field.

(c) Copyright A. Gerbessiotis. All rights reserved.

B. Information recorded in inverted lists.

B1. Doclist (i.e. documents). The simplest form of an inverted list stores just the documents than contain a given index-term and no other information. We can think of this information as if storing the 1s of an incidence vector by storing for each one a reference (`docID`) to the relevant document. If doclists are maintained in sorted order we can find the intersection of two of them or their union (and thus perform a bitwise OR and AND) in time proportional to the longest of the two lists. **In short there is a single posting per document and the posting contains only a pointer (information).**

B2. Counts. For each document listed in the inverted list of a given index-term, a count records the number of times the index-term occurs in that document. **A single posting per document is available with the posting containing pointer and count information.**

B3. Positions. This expands the information available through **Counts**. The number of postings for a particular document depends on the counts information and is as much as the counts field indicates. Each posting contains a pointer that includes document information plus a position within the document. Such position information can be in the form of a character offset (rarely used), a word offset, or a block offset if a document is split into blocks of coarser granularity (a block can be a number of consecutive characters or bytes, or a paragraph, or a section, or a chapter). The offset is from the beginning of the document. Note that a maximum size of an offset can be established. For example in Google (1997) a word offset can grow up to 4094 with 4095 used for words with offset greater than 4094.

B4. Fields and Extents. Emails contain sender information and a subject line. HTML documents contains title information. Information located in those fields can be associated with these **fields** that are sections of the document that carry some specific semantic meaning. An **extent** is a consecutive area (i.e. consecutive words) of a document storing some specific information possibly about a field. For example an HTML title is stored between `<TITLE> and </TITLE>`.

B5. Context Information. It records information about the structure around the term. This can include HTML tag information, font-size, capitalization that can be used in scoring processing (i.e. query-related document ranking).

B6. Ordering of an inverted list. The usual ordering of a list is by document reference (i.e. `docID`). It is possible however, that a score can be assigned to a particular document. Then it might make more sense to order documents based on this score.

(c) Copyright A. Gerbessiotis. All rights reserved.

Example of building one

We give a more comprehensive example right now that shows all the steps performed from tokenization down to index creation.

Let the three documents of this example be the ones indicated. The whole indexing process consists of the following rounds.

- **1. Tokenization and 2. Linguistic Analysis.** This phase first generates for each document a stream of tokens with Positions information in the form of `(wordID,docID,offset)` tuples. The tuples are then sorted based on `wordID` and secondarily (or by preserving previous order) based on `docID, offset`.
- **3. Forming and Building the index.** The sorted stream is processed to determine (i) the number and list of unique words, (ii) `Ndocs`, the number of documents containing a given `wordID`, (iii) `Nhits` the number of occurrences of a `wordID` in each one of the `Ndocs` documents. Parsing sequentially the left part of the next figure can collect this information. At the same time `Nhits, Hits` information is incorporated and depicted in the right part. This is the first step of building the index i.e. forming the Vocabulary and the inverted lists. Immediately afterwards, the vocabulary is built, and inverted list table entries are coalesced. By going through the table of the right part of the figure we can decide the number of lines in the table of inverted lists and build the `LocP` pointers to that table.

```
  Document 1.              Document 2.             Document 3.
  ------------------------------------------------------------
data structures            data structures         algorithms in java
and algorithms             and their
in java                    algorithms
```

Example of building one (continued)

```
       Phases 1 and 2.
-Tokenization
-No linguistic analysis
 or filtering
```

wordID	(docID,offset)
data	(1,1)
structures	(1,2)
and	(1,3)
algorithms	(1,4)
in	(1,5)
java	(1,6)
data	(2,1)
structures	(2,2)
and	(2,3)
their	(2,4)
algorithms	(2,5)
algorithms	(3,1)
in	(3,2)
java	(3,3)

```
Phases 1 and 2.
Sort the output based
on
 - wordID
maintaining previous order based on
 - docID
 - word offset
(DocID relative order remains)
```

wordID	(docID,offset)
algorithms	(1,4)
algorithms	(2,5)
algorithms	(3,1)
and	(1,3)
and	(2,3)
data	(1,1)
data	(2,1)
in	(1,5)
in	(3,2)
java	(1,6)
java	(3,3)
structures	(1,2)
structures	(2,2)
their	(2,4)

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Example of building one (continued)

Phase 3. Forming the index.

wordID	docID	wordID	docID	Nhits	Hits
algorithms	(1,4)	algorithms	1	1	4
algorithms	(2,5)	algorithms	2	1	5
algorithms	(3,1)	algorithms	3	1	1
and	(1,3)				
and	(2,3)	Similarly.			
data	(1,1)	(details			
data	(2,1)	omitted)			
in	(1,5)				
in	(3,2)				
java	(1,6)				
java	(3,3)				
structures	(1,2)				
structures	(2,2)	structures	2	1	2
their	(2,4)	their	2	1	4

Phase 3. Building the index: Vocabulary and Inverted Lists.

Vocabulary / Inverted List Table

wordID	Ndocs	LocP		docID	Nhits	Hits
algorithms	3	0	0	1	1	4
and	2	3	1	2	1	5
data	2	5	2	3	1	1
in	2	7	3	1	1	3
java	2	9	4	2	1	3
structures	2	11	5	1	1	1
their	1	13	6	2	1	1
			7	1	1	5
			8	3	1	2
			9	1	1	6
			10	3	1	3
			11	1	1	2
			12	2	1	2
			13	2	1	4

(c) Copyright A. Gerbessiotis. All rights reserved.

We show how simple queries can be satisfied.

Let us have a very simple conjunction of three terms `algorithms AND data AND structures`. In this example we use the standard convention that a capitalized `AND` is the conjunction operator (logical AND) rather than index-term `and`. The following steps are performed to generate the unified hitlists for the three index-terms.

- The `wordID` is found for the index-term and its position located in the vocabulary. The number of documents `Ndocs` that contain it are then located and the corresponding lines of the table of inverted lists are then accessed using `LocP`, i.e. lines `LocP, ... , LocP+Ndocs-1` will be retrieved.
- For each line of the table of inverted lists retrieved, the `Nhits` is retrieved to determine the structure of `Hits`. The hitlist `Hits` is then decompressed
- The lists for each line are concatenated (unified) so that each term gives rise to one such list.
- Further processing is performed to satisfy the conjunction (this example) or disjunction as applicable.

```
 algorithms    3    0   --> 0  1  1  4   -->  (1,4)   -->  (1,4)(2,5)(3,1)
                            1  2  1  5   -->  (2,5)
                            2  3  1  1   -->  (3,1)
 data          2    5   --> 5  1  1  1   -->  (1,1)   -->  (1,1)(2,1)
                            6  2  1  1   -->  (2,1)
 structures    2   11   --> 11 1  1  2   -->  (1,2)  --->  (1,2)(2,2)
                            12 2  1  2   -->  (2,2)

algorithms  (1,4) (2,5) (3,1)
data        (1,1) (2,1)
structures  (1,2) (2,2)
```

Having retrieved the hitlists for the query index-terms we are ready to satisfy the query. The following method in satisfying a simple conjunction or disjunction with two index terms gives an answer utilizing only `docID` information and ignores the additional attributes such as number of occurrences `Nhits` in a document or offset information. Thus the **scoring** mechanism that determines whether a document is better than another one is very primitive: all documents are the same no matter how many times contain a given index-term. One can improve up on this in a variety of ways: using `Nhits` information or also taking into consideration of the proximity of index terms as expressed by the `offset` attribute stored in `Hits`.

- **Naive Scoring Mechanism**
  - **Conjunction:** `x AND y`**.** For a conjunction we need to find all documents that contain both index-terms (say x and y). This is thus equivalent to finding the intersection of the hitlists using only `docID` information but ignoring offsets.
  - **Disjunction:** `x OR y`**.** For a disjunction we need to find all documents that contain one or the other (or both of the index-terms). This is thus equivalent to finding the union of the hitlists using only `docID` information but ignoring offsets.
  - **Negation:** `NOT x` **or** `-x`. We need to find all the documents that do not contain x. For this we subtract from the documents of the corpus those containing x. This is the set theoretic complement operation.

```
algorithms  (1,4) (2,5) (3,1)   Naive scoring mechanism  -->  1,2,3
data        (1,1) (2,1)         (retain docID info           1,2
structures  (1,2) (2,2)          filter out offsets)         1,2

                                                              _
Query is   :    algorithms AND data AND structures     Note :  | |  is the intersection symbol (upsidedown U)

((algorithms AND data) AND structures)          (algorithms AND (data AND structures))
                 _                                            _
 temp1: {1,2,3} | | {1,2} = {1,2}         temp3 = {1,2}     | | {1,2} = {1,2}
                 _                                            _
 temp2: temp1   | | {1,2} = {1,2}         temp4 = {1,2,3}   | | temp3 = {1,2}
```

Disjunction and Conjunction resolution through merging: 2 terms/sets

Finding the Intersection and Union by merging lists. In order to find the intersection of two sets or two sequences quickly, the two sequences must be in sorted order (increasing/non-decreasing order of `docID`).

Conjunction by intersection through merging. Disjunction by union though merging.

Say we are given two sorted (doc)lists $A = \{1, 2, 3\}$ and $B = \{1, 2\}$ of `docID`s for terms A and B. We write the elements of the two lists with subscripts to denote the set/term to which they belong. (The incorporation of subscripts is just for illustration and can be avoided.)
$A = \{1_A, 2_A, 3_A\}$ and
$B = \{1_B, 2_B\}$.

The merging of A and B would generate as output $\{1_A, 1_B, 2_A, 2_B, 3_A\}$.

Intersection of A and B. With a linear scan we can (i) eliminate non-duplicates, (ii) keep one copy of duplicates, and (iii) remove subscripts so that we finally get $\{1, 2\}$. This is the intersection of A and B.

Union of A and B. For the union, we only keep one copy of duplicates without eliminating non-duplicates and then remove subscripts. Thus we get $\{1, 2, 3\}$. This is the union of A and B.

The running time of merging is $m + n - 1$ where m is the number of elements of A and n the number of elements of B. (And this implies A and B are already sorted.)

More complex queries involving `NOT`. What if the query involves a `NOT`? What is the result of `NOT data`? It is all the documents in the collection that do not include `data` that is for our example this is $\{3\}$. Thus `algorithms AND NOT data` requires the intersections of the hitlist of `algorithms` with the hitlist not of `data` but the complement of `data` i.e. all the documents of the collection minus those containing `data`. Thus the set theoretic `complement` operation allows for the implementation of operator `NOT`.

Conjunction by intersection by modified/optimized merging. Disjunction by intersection by modified/optimized merging. On page 23 we show how to optimize the intersection/union without creating a merged output.

(c) Copyright A. Gerbessiotis. All rights reserved.

Disjunction and Conjunction resolution through merging: 3 or more terms/sets

More than two operands in a query or more than two sets to merge. If we have more than two terms in a conjunction then the evaluation can be performed in a variety of ways. For example we can either evaluate the first two terms and the result with the third (i.e. find the intersection of the hitlists of the first two terms and then the intersection of the result with the hitlist of the third term) or work out the evaluation of the second and third terms and then combine them with the first one. `Is there a difference between one or the other schedule? Yes, the total (operation) cost might be different! (think of nested matrix multiplication)`

Associative Operators. An operator $\oplus$ is associative if and only for all a, b, c we have $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. Operators such as $+, *$ and `AND, OR, MIN,MAX` are all associative. If the cost of applying an operator is unit, then whether $a \oplus (b \oplus c)$ is performed or $(a \oplus b) \oplus c$ does not matter: 2 operations will be performed at a cost of two units. The operators Intersection/Union through merging however are more costly: the cost depends on the length of hitlists.

Example 1. Merge-Delay-Prune. Merging three sets A and B and C of sizes n, $2n$ and $3n$ with all elements distinct from each other.. Say we merge first A and B and then the result with C. The cost of the first merging is $n + (2n) - 1 = 3n - 1$ and results into a set of $3n$ elements. Then merging this set with C will cost $3n + 3n - 1 = 6n - 1$ and give a set of no more than $n + 2n + 3n = 6n$ elements. The total cost of the two merging operations is $3n - 1 + 6n - 1 = 9n - 2$.

If however we first merge B and C to get a set of size $2n + 3n = 5n$, the cost of this merging will be $2n + 3n - 1 = 5n - 1$. The resulting set of $5n$ elements if merged with A will still give $n + 5n = 6n$ elements but this last merging step will cost us $5n + n - 1 = 6n - 1$. The total cost for merging is then $(5n - 1) + (6n - 1) = 11n - 2$, which is more than 20% more expensive than the first schedule.

Rule 1. Merge the shorter lists first!

Example 2. (Merge-and-Prune) Intersection of three sets A and B and C of sizes n, $2n$ and $3n$. Although intersection can be implemented through merging, we know something else about intersection: the size of the result CAN BE NO LARGER THAN THE SIZE OF THE SMALLEST SET. In fact it will be a subset of the smallest set, i.e. A in our case. This alone dictates that not only should we process smaller sets first, but do the pruning (page 18 example involving duplicate elimination) early on! (In that case merging A, B at a cost of $n + 2n - 1 = 3n - 1$ and then performing the duplicate elimination reduces the intersection of A, B to size no more than n, and thus the second intersection will merge this latter set with C in $n + 3n - 1 = 4n - 1$ operations for a total cost of $(3n - 1) + (4n - 1) = 7n - 2$; the alternative of first doing both merges and then the pruning out is way too costly as Example 1 indicates.)

Rule 2. Merge-and-Prune is better than Merge-Delay-Prune.

(c) Copyright A. Gerbessiotis. All rights reserved.

Intersection in reality. The intersection algorithm for `AND` and union algorithm for `OR` can be further optimized to perform the reduction of duplicates and the elimination or not of non-duplicates during the scanning of the two hitlists rather than in a separate scan of the merged list output. Thus neither `Merge-Delay-Prune` not `Merge-and-Prune` are optimal. We call the new approach (optimized) `Intersect_for_AND`.

Thus for the sets $A = \{1, 2, 3\}$ and $B = \{1, 2\}$ the merged output $\{1_A, 1_B, 2_A, 2_B, 3_A\}$ will never be built but instead $\{1, 2\}$ will only be output directly. The pseudocode follows.

```
 Intersect_for_AND (L,R)
0.   i=1;
1.   s=Ndocs(L);      % length of left  sequence L is s
2.   t=Ndocs(R);      % length of right sequence R is t
3.   S=1;              % Runs through L
4.   T=1;              % Runs through R
5.   while i <=  (s+t) % Build output one element at a time
6.     if  S >  s
7.        return(AND,i)     % Size of AND is i
8.     elseif  T > t
9.           return(AND,i) % Size of AND is i
10.    elseif L(S) <  R(T)
11.          S=S+1;        % Advance L by one
12.    elseif L(S) >  R(T)
13.          T=T+1;        % Advance R by one
14.    else                 % L(S)==R(T)
15.          AND(i)=L(S);S=S+1;T=T+1;i=i+1;
16.    end
17.  end
```

In other words while we scan one element $l = L(S)$ from one list and one element $r = R(T)$ from the other list if

- (i) $l < r$ we advance to the next element of the left list,
- (ii) $l > r$ we advance to the next element of the right list,
- (iii) $l == r$ we have determined an intersection element and store it in array `AND`.

Thus all this activity, can be determined from the initial stream of tuples, by performing a sequence of **sorting operations** followed by a sequence of **scanning operations** that might involve duplicate elimination or counting of duplicate entries (where duplicates are `wordID`s, `docID`s or pairs of `(wordID,docID)`s). (Consider Google in 1997 in which the sorter operated on the forward index stored in the barrels to generate the inverted index to be stored in the barrels as well.)

All those tasks involve streams of tuples in the billions or trillions. It is rather impossible for all these operations to be performed by a single computer. They have to be performed **in parallel** by a collection (i.e. cluster) of several (tens or hundreds or thousands) machines. Both **sorting** and **scanning** operations are amenable to efficient parallelization.

Whether to count or not, eliminate or not, and how to count (duplicate or unique occurrence counting) involves a `mapping` of the original stream into a `mapped` set.

Thus a **MAP** operation is essential. The end result of the `mapping` is a **Reduction** operation, or **REDUCE** that will eliminate duplicates or count tuples in a given way as determined by the `MAP` operation. In addition if several machines are being used for this task, the `MAP` operation also works so that the tuples are distributed evenly to the machines of the collection based on the mapped values. Thus tuples with a given `(wordID,docID)` value might be destined for machine X instead of machine Y, and thus the `REDUCE` operation will be applied to them then (after they move to that machine X). Similarly the `REDUCE` operation will also collect partial results per machine X, to generate global results per collection rather than per machine that will be then sent to the `query server or GWS`.

This is the fundamental idea behind Google's **MAPREDUCE** system: a convenient way to automate several similar-looking repetitive and time-consuming tasks, and also the full and efficient parallelization of this activity due to the sizable amount of information involved. (And by parallelization we mean the splitting of data evenly among the machines of the cluster, the transmission of data to the machines based on this splitting, the schedule and synchronization of the computations involved, the transmission and collection of partial/local results, and finally the generation of global results with their dispatch to one machine, if they are small in size, or more than one machines otherwise.)

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

Information retrieval systems assume that the semantics of the document and of the user query can be modeled by a set of index terms. This approach may be problematic since

- A user request might get prepared using an imprecise space of index terms (eg. all the words in a document) rather than a clearly/precisely defined such set.
- **Food for Thought:** What if we search `A is better than B` or `B is better than A` by performing a keyword search? Will the results differ? How relevant will they be to one or the other query?
- Users are not trained to properly form their queries.
- Which documents of the output are relevant and which are not ? **Ranking**
  - Ranking operates on premises regarding the notion of document relevance.
  - Distinct sets of premises yield distinct information retrieval models
  - The IR model one adopts determines what is relevant and what is not.

There are three classical information retrieval models: boolean, vector, and probabilistic.

- In the **boolean model**, documents and queries are represented as sets of index terms, i.e. the model is *set-theoretic*.
- In the **vector model**, documents and queries are represented as vectors in t-dimensional space i.e. the model is **algebraic**.
- In the **probabilistic model**, the modeling framework is based on probabilities i.e. it is probabilistic.

(c) Copyright A. Gerbessiotis. All rights reserved.

- **F** is the **modeling framework** that we will be using at any time.
- **D** is a collection (also known as corpus) of **n** documents. $\mathbf{d_j}$ is a **document** of the collection.
- **Q** is a set of query requests. Since we rank results of individual queries we will denote by q one element of **Q** or simply consider **Q** to have only one element and that element will be **query** q.
- $\mathbf{k_i}$ is an **index term** $1 \leq i \leq t$, where **t** is the **number of terms**.
- $\mathbf{n_i}$ is the number of documents in which k_i appears i.e. the **document frequency** of term k_i.
- $\mathbf{R(q, d_j)}$ is a ranking function which associates a real number with a query q and document d_j. The rank might be based entirely on the similarity between the query q and the document d_j i.e. $s(q, d_j)$.

*** $\mathbf{f_{ij}}$ is the raw frequency of k_i in d_j. This is the **term frequency** or **tf**-factor.

- $\mathbf{F_{ij}}$ is the normalized frequency of k_i in d_j i.e. $F_{ij} = \frac{f_{ij}}{\max\limits_{l} f_{lj}}$.

*** $\mathbf{idf_i}$ is the **inverse document frequency** for k_i i.e. $\mathbf{idf_i} = \log \frac{n}{n_i}$ or $\mathbf{idf_i} = \log\left(1 + \frac{n}{n_i}\right)$. It gives high values for rare words, and low values for frequent ones.

- $\mathbf{w_{ij}}$, $\mathbf{w_{iq}}$ are the **document-term weight** (denoted by w_{ij}) and the **query-term weight** (denoted by w_{iq}) respectively and express the weight of index term k_i in document d_j and query q respectively. $w_{ij} = F_{ij} \times idf_i$.
  - A normalized term is given by $w_{ij} = \dfrac{f_{ij}\mathbf{idf_i}}{\sqrt{\sum\limits_{i} f_{ij}^2 \mathbf{idf_i^2}}}$
  - A variant is defined (Salton and Buckley) as $w_{iq} = \left(0.5 + \dfrac{0.5 * F_{iq}}{\max\limits_{l} f_{lj}}\right) \times \mathbf{idf_i}$, for q a query request.
- $\mathbf{r_{ij}}$, $\mathbf{r_{iq}}$ are the **relative term frequency** and the **relative query frequency** and express the relative frequency of index term k_i in document d_j and query q respectively. (One can consider $r_{iq} = 1$, and r_{ij} a value dependent on either F_{ij} or $(1 + \log f_{ij})$.)

- **Intra-cluster similarity** Determine the features that best describe the objects of set A. A measure of similarity is **tf**.
- **Inter-cluster dissimilarity** Determine the features that best distinguish the objects of set A from those of the collection C. A measure of dissimilarity is **idf**.

A set of values for w_{ij} and w_{iq}. In general, one can use the following set of values

$$\begin{aligned} \mathbf{idf_i} &= \log\left(1+\frac{n}{n_i}\right) \quad , \\ \mathbf{r_{ij}} = (1+\log f_{ij}) \quad &, \quad \mathbf{r_{iq}} = 1 \\ \mathbf{w_{ij}} = r_{ij} \quad &, \quad \mathbf{w_{iq}} = r_{iq}\mathbf{idf_i} \end{aligned} \tag{1}$$

This way, for a query q and document d_j the similarity $s(q,d_j)$ can be expressed by a formula of the form

$$s(q,d_j) = \frac{\sum_i w_{ij} w_{iq}}{\sum_i f_{ij}} \tag{2}$$

where the denominator is the length of document d_j in index-terms (not characters and also not necessarily in words). When we discuss the vector model in the sections to follow, a more thorough investigation will take place.

Option 1 for rank.

$$R(q,d_j) = s(q,d_j).$$

Option 2 for rank.

$$R(q,d_j) = s(q,d_j) + \textbf{Other factors}.$$

(c) Copyright A. Gerbessiotis. All rights reserved.

Definitions

- Documents are also represented as vectors: $d_j = (w_{1j}, \ldots, w_{ij}, \ldots, w_{tj})$.
- Queries are also vectors $q = (w_{1q}, \ldots, w_{iq}, \ldots, w_{tq})$ i.e. become "short documents" themselves.
- The similarity between q and d_j can be defined as follows.
  - **Cosine of the angle**. $s(q, d_j) = \frac{q \cdot d_j}{|d_j| \times |q|} = \frac{\sum_i w_{ij} w_{iq}}{\sqrt{\sum_i w_{ij}^2} \sqrt{\sum_i w_{iq}^2}}$. This is a normalized similarity metric.
  - **Euclidean Distance**. $s(q, d_j) = \sqrt{\sum_i (w_{ij} - w_{iq})^2}$.
  - **Manhattan Distance**. $s(q, d_j) = \sum_i |w_{ij} - w_{iq}|$.
  - **Dot-Product distance**. $s(q, d_j) = \sum_i w_{ij} * w_{iq}$. (The one used in the previous page.)
- Non-binary weights provide for partial matches.
- Degree of similarity is possible.
- Ranking is feasible.
- How to answer a query. For a query q
  - For all documents d_j determine similarity $s(q, d_j)$.
  - Rank documents based on similarity measure.
  - For cosine similarity, best angle must be as close to zero as possible i.e. the best metric should be very close to 1 (since cos0=1).

(c) Copyright A. Gerbessiotis. All rights reserved.

Discussion

- Keyword frequencies favors frequent words which might not convey enough information about document, tf-factor should be overemphasized. F_{ij} become important but not much.
- Inter document dissimilarity i.e. the idf-factor must also be used.
- Therefore $w_{ij} = F_{ij} * idf_i$. Also called **tf-idf** scheme/weights.
- If one wants to give more emphasis on rare words, use only idf_i and disregard the other contribution (F).
- **Advantages.** Good ranking (term weighing, partial matching, cosine similarity).
- **Disadvantages.** Assumes terms are independent; does not handle synonyms. Query weighing may not reflect relevance.
  - **Dot-product**. Favors longer over shorter documents. Similarity not scaled.

(c) Copyright A. Gerbessiotis. All rights reserved.

Algorithmic Issues

- To determine the similarity between a query and a document collection we need to evaluate $s(q, d_j)$ for all j. This is computationally expensive.
- An alternative way is to use the inverted index. In the inverted index, in an occurrence list, we maintain along with the `docID` also a `docW` (number of all words in the document) that can be computed from `Nhits` (to determine f_{ij}, F_{ij}) and `Ndocs` (to determine idf_i).
- Therefore we can compute the similarity $s(q, d_j)$ for query q for all documents j in the collection as follows.

```
// Compute similarity for query q for all n documents of the collection
         s(query q) //
1. for(j=0;j < n ; j++)  s(q,dj)=0.0;     // Initialize a counter for each document.
2. for(i=0;i < q.length(); i++) {         // Go through each term of the query one by one
3.  ki = q[i];                            // Let the i-th term of query q be ki
4.  Location= LocateInInvertedIndex(ki);  // Locate the entry for ki in the Vocabulary
5.  Ndocs = InvertedIndex[Location].Ndocs;// Locate the Ndocs for term ki
5.   for(k=0;k<Ndocs;k++) {               // Go through the Ndocs documents that contain ki
6.      DocIndex = LocP+k;                // The k-th document is in location LogP+k of the
                                          // occurrence list
        // Compute the contribution to the document/term pair in s(q,DocID);
7.      s(q,OccurrenceList[DocIndex].DocID) = OccurrenceList[Docindex].Nhits *
8.                                          log(N/InvertedIndex[Location].Ndocs);
9.    }//and repeat for all docs of ki.In line 7 we use fij not Fij (i.e. no normalization)
10. } // and repeat for all query terms of q
```

- The running time instead of being proportional to $n \times t$ it is proportional to n (number of documents) and $\sum_i n_i$ where n_i is the number of documents in which k_i appears.

(c) Copyright A. Gerbessiotis. All rights reserved.

Definitions

There are many models on ranking documents and web-pages: five of them them were described earlier. In order to compare them, one need to be able to evaluate and quantify their performance in some way. For the evaluation of any model, two important measures of effectiveness are **precision** and `recall`.

Precision is the fraction of the retrieved documents that are relevant to the total number of retrieved documents. This fraction of retrieved documents relevant to the query as it was expressed by the user. In general, one want to maximize precision as much as possible.

```
                  # of retrieved documents that are relevant
Precision = P = ----------------------------------------------
                            # of retrieved documents
```

Recall is the fraction of the relevant documents that are retrieved. The recall is a non-decreasing (i.e. increasing) function. One wants to maximize recall.

```
                  # of retrieved documents that are relevant
   Recall= R = ----------------------------------------------
                            # of relevant  documents
```

Example. Say that 40 documents are relevant to a query q. The model tested retrieved for query q 50 documents; out of those 50 documents only 30 are relevant. The precision is 30 over 50 i.e. $P_{50} = 30/50 = 0.6$. The recall is 30 over 40 i.e. $R_{50} = 30/40 = 0.75$

Note. One "naive" way to get a high recall rate is to output all the documents of the collection. The recall rate would be 1 (i.e. 100%). Yet the precision for this case will be very very low, since any non-relevant document will also be output.

Note. Precision usually decreases in a well-defined/structured system.

(c) Copyright A. Gerbessiotis. All rights reserved.

1. Set of all documents in the corpus. Consider a query request q on a corpus of n documents.

2. Set A of relevant documents. The set of relevant documents to the query is A and its cardinality is a.

3. Set B of retrieved documents. An information retrieval policy retrieved the documents of set B. Its cardinality is b.

4. Set T of retrieved documents that are relevant. The set $T = A \cap B$ contains all the retrieved documents that are relevant. Let its cardinality be t.

The top-left element of the table (disregarding the TOTAL rows and columns) shown below shows the cardinality of T the number of retrieved documents that are relevant. Then $b - t$ is the number of retrieved documents that are not relevant (i.e. they irrelevant) to the query. This the top-right element of the table below.

The number of relevant documents not retrieved is $a - t$ i.e. the difference between the number of all relevant documents in A minus those relevant documents retrieved in B and formed T. This is the bottom-left element of the table.

What needs to be confirmed is the number of irrelevant document not retrieved. The collection has n document of which a are relevant. Thus the number of irrelevant documents is $n - a$. Those retrieved are $b - t$, thus those not retrieved are $n - a - b + t$.

The ratio $(b - t)/(n - a)$ is called the **fallout** and gives the fraction of the irrelevant but retrieved documents over all the irrelevant documents.

	RELEVANT	IRRELEVANT	(TOTAL)
RETRIEVED	t	b-t	b
NOT RETRIEVED	a-t	n-a-b+t	n-b
(TOTAL)	a	n-a	

PRECISION P = t / b The first-row information is used to derive P.
RECALL R = t / a The first-column information is used to derive R.
A= set of relevant documents B=set of retrieved documents T= relevant documents retrieved

Example. Say we have the sequence of documents (in **bold-face** are the relevant documents) shown.

$$\mathbf{d_1}d_2d_3d_4\mathbf{d_5}d_6d_7d_8\mathbf{d_9}\mathbf{d_{10}}d_{11}d_{12}d_{13}d_{14}d_{15}d_{16}d_{17}\mathbf{d_{18}}d_{19}d_{20}$$

Suppose we measure precision and recall every time a document is encountered left-to-right. (One can use percentages to express recall and precision as we do in the example on the next page.

The precision falls from 100 to $40, 33\%$, then goes up to 40% and eventually stabilizes at 25%.

The recall starts with 20% and eventually grows to 100% assuming that the relevant documents are only the ones listed i.e. $d_1, d_5, d_9, d_{10}, d_{18}$.

Thus one can draw a curve for precision and separately for recall during this experiment.

Sometimes we prefer to use the Figure were recall and precisions figures are computed after every document is considered in turn. We can also draw a smaller table such as in which recall and precision figures are derived at incremental recall rates: in our example we used a 6-step 20% increment table with rows at `0% , 20% , 40% , 60 % , 80% , 100 %` recall rates. One could have used instead (if it was possible) an 11-step 10% increment table. An interpolated precision can then be derived, which is the maximal precision value encountered in that line or in lines below the given recall level. For example at 20% recall level the interpolated precision is the maximum of all precisions. At 60% it is the maximum of $33\%, 40\%, 27\%$. We can then compute a full-point average of interpolated precision rates. Thus a 6-point average gives us $(100 + 100 + 40 + 40 + 40 + 27)/6 = 57\%$. A 3-point average can usually be drawn at recall rates of $20\%, 50\%, 80\%$. In our case, the 50% recall rate uses a 40% interpolated preceision figure which was taken either from the 40% or 60% recall rate to determine a 3-point effectiveness. The `3-point, or 11-point, or full-point` average of interpolated precisions are known as `3-point` **effectiveness**, `11-point` **effectiveness**, etc.

(c) Copyright A. Gerbessiotis. All rights reserved.

Example continued

r	dr	Relevant	Rr (recall)	Pr (precision)
1.	d1	yes	20%	100%
2.	d2		20%	50%
3.	d3		20%	33%
4.	d4		20%	25%
5.	d5	yes	40%	40%
6.	d6		40%	33%
7.	d7		40%	28%
8.	d8		40%	25%
9.	d9	yes	60%	33%
10.	d10	yes	80%	40%
11.	d11		80%	36%
12.	d12		80%	33%
13.	d13		80%	30%
14.	d14		80%	28%
15.	d15		80%	26%
16.	d16		80%	25%
17.	d17		80%	23%
18.	d18	yes	100%	27%
19.	d19		100%	26%
20.	d20		100%	25%

(a)

Recall	Precision	Interpolated precision
0%	-	100%
20%	100%	100%
40%	40%	40%
60%	33%	40%
80%	40%	40%
100%	27%	27%
3-point average		100+40+40/3 =60%
6-point average		57%

(b)

(c) Copyright A. Gerbessiotis. All rights reserved.

Another interesting metric is the **precision-recall** curve that combines data from the two previous curves/metrics but is used in evaluating multiple queries. For a set of test queries we can measure the precision figures for various recall levels as before such as at $0\%, 10\%, 20\%, \ldots, 100\%$ (i.e. 11 levels or alternately 6 levels as before) and then evaluate the retrieval process by using the sum

$$P(r) = \frac{\sum_{i=1}^{n_q} P_i(r)}{n_q}$$

where n_q is the number of queries used for the evaluation, $P_i(r)$ is the precision figure of the i-th query at recall level r. Interpolation might be used to determine precision figures at given recall levels that might fall in between the step values.

A better approach is to use a **single figure description** instead of a curve for precision-recall relationships. This can be done as follows. (One way to do this is by providing 3-point or 6-point or 11-point effectiveness figures.) Alternately we could do the following.

Step 1. Compute the precision figure after each new relevant document is observed, and compute the average precision.

Step 2. Find the precision at the q-th position in the ranking, i.e. find the relevant documents among the first q documents and then compute the precision.

Step 3. Another interesting measure combines P and R through the weighed harmonic mean WHM which is defined as

$$\mathbf{WHM} = \frac{1}{a\frac{1}{P} + (1-a)\frac{1}{R}}$$

where a is a weight. A choice of a is $a = 1/2$ (see page 310 of the textbook).

A reason of preferring the harmonic mean over say the arithmetic mean can be shown by the following example. Say we return all the documents in the collection as a result to a query. Then recall is 1 (100%) but precision is close to 0. The arithmetic mean (average) is 0.5 (or 50%), whereas the harmonic mean will also be close to 0.

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

For web-based documents that are in HTML format, one can formulate weight factors that reflect the position of the index terms in the HTML text. The Webor Method (Cutler et. al.) partitions HTML tags into ordered classes. It then determined approximate percentages of the distribution of term occurrences among a set of six ordered classes. These are listed below.

- **title** (eg. <TITLE>), Distribution: $< 1\%$.
- **anchor** (eg <A>), Distribution: $< 3\%$.
- **header** (eg. <H1> to <H6>), Distribution: $\approx 4\%$.
- **list** (eg. <DL>,<OL>,<UL>), Distribution: $\approx 5\%$.
- **strong** (eg. <STRONG>, <B>, <EM>, <I>, <U>), Distribution: 10%.
- **plain** (eg. none of the above) Distribution: $< 75\%$.

Then, instead of using a single tf-value f_{ij} or F_{ij} for a term to describe w_{ij}, Webor uses a tf-vector (tfv) such as

$$\mathbf{tfv} = (tf_1, tf_2, tf_3, tf_4, tf_5, tf_6)$$

where tf_i is the tf-term for occurrences of the index term in a class i tag in the document. For each element of the vector different **importance values (iv)** are then assigned. Let iv_i be the importance values for the i-th class.

The new term-frequency is then computed by the combination of the tf-values of the classes weighed appropriately.

$$\sum_i tf_i \cdot iv_i.$$

Some conclusions based on experimentation: Anchors and strong/emphasized text are important, header information is less, title is slightly just above plain text and listings.

(c) Copyright A. Gerbessiotis. All rights reserved.

The relevance of a query to a document (e.g. a web-page) is established by first computing the function $s(q, d_j)$ that determines the similarity between query q and document d_j.

This similarity measure is based solely on information retrieval-based knowledge (e.g. the searchable contents of the document). Search engines (or web-searching methods in general) do not rely on this measure exclusively to rank a web-page (e.g. document d_j).

They also use additional information such as the link structure of d_j. The link structure relates to information such as:

- (a) what documents/pages are pointed by d_j and are these pages important (e.g. authoritative),
- (b) what documents/pages point to d_j and are these pages important (e.g. hubs of information such as a directory page).

All such link information can be made to contribute a separate component to the rank of d_j or a web-page in general.

Thus the **rank** of d_j relative to query q denoted by $R(q, d_j)$ is a function that will represent the rank of a document/web-page. One component of $R(q, d_j)$ is $s(q, d_j)$.

One of the first attempts to use linkage information was the **vector spread activation model** by Yuwono.

Vector spread activation model. The ranking score $R(q, d_j)$ is the sum of the similarity $s(q, d_j)$ of d_j plus a portion of the similarity of each document that points to d_j. This is under the assumption that if relevant documents point to d_j, then d_j should also be relevant. Let $l(i, j)$ be 1 or 0 based on whether document d_i points to d_j or not. Then

$$R(q, d_j) = s(q, d_j) + \beta \sum_i R(q, d_i) l(i, j)$$

where β is a user defined parameter (e.g. $\beta = 0.2$).

References:

1. M. Cutler, Y. Shih, and W. Meng: Using the Structures of HTML Documents to Improve Retrieval. Usenix Symposium on Internet Technologies and Systems (USITS'97) . pp.241-251, Monterey, California, December 1997.

2. M. Cutler, H. Deng, S. Manicaan, and W. Meng: A New Study on Using HTML Structures to Improve Retrieval. Eleventh IEEE Conference on Tools with Artificial Intelligence (ICTAI'99) , Chicago, November 1999.