

*This homework is worth 50 points as shown. By the deadline, email your solution to the TA or ask the main office of the CS building to put a copy in the professor's mailbox.*

### **A. Text statistics (15 points)**

Section 4.2.2 of the CMS textbook discusses a method for estimating the size of a corpus based on statistics returned in response to queries. Specifically, suppose the probability of seeing a document containing a word  $w$  is  $df_w/N$  and of seeing a word  $v$  is  $df_v/N$  where  $df_w$  is the number of documents containin the word  $w$  and  $N$  is the total number of documents in the collection. If  $w$  and  $v$  occur independently, then the probability of seeing both  $w$  and  $v$  is the product, but it is also  $df_{w+v}/N$ . That means that,

$$\frac{df_{w+v}}{N} = \frac{df_w}{N} \cdot \frac{df_v}{N} \quad \rightarrow \quad N = \frac{df_w df_v}{df_{w+v}}$$

Use that information to estimate the size of the corpus indexed by Google, Yahoo, and Bing. Use several different pairs of words to make your estimate.

There is no solution to this problem. An interesting observation was that some people got different counts for the same word from the same search engine. What does that tell us about the search engines?

### **B. Inverted list (15 points)**

Term-at-a-time evaluation of queries processes a single inverted list at a time. If it were possible to skip some of the inverted lists in some cases, it would necessarily reduce the time it took to process that query.

Suppose that all we care about is that the top  $k$  documents returned are the correct top  $k$ , but we aren't concerned whether we have precisely the right ranking. That is, we want to assign approximate scores for ranking documents such that any document that would be in the top  $k$  by the true score is also in the top  $k$  by the approximate score.

Propose a variation on the term-at-a-time approach that stops updating scores but achieves that goal. Describe why your approach may result in less processing. You may modify the inverted list structure by adding fields or reordering entries if it is valuable to do so, but you must explain what you are requiring. Be sure you explain how this achieves our goal of getting the top  $k$  correctly in the top  $k$  ranks.

Your approach almost certainly does not *always* reduce the amount of time it takes to process queries? Under what condition will it not? Support your answer (in a sentence or two at most).

How would you modify your algorithm if you still wanted to minimize effort but wanted to ensure that document order was correct in the top  $k$ ? Note that you don't care about order outside of the top  $k$  documents.

We present two possibilities. In each case, consider a retrieval model whose scoring function is of the form:

$$f(Q,D) = \sum_{t \in Q \cap D} w_{t,q} \times w_{t,d}$$

where,

$w_{t,q}$  is a document independent weight of term  $t$  (such as  $P(t|Q)$ ), and  
 $w_{t,d}$  is the document weight of term  $t$  (such as  $\log(P(t|D))$ ).

*Index Time Changes:*

1. When creating the index, store the document weights of the term, instead of just storing the frequencies.
2. Furthermore, store the largest document weight as the *maximum contribution* for each term.

*Modified Term-at-a-Time Algorithm: MOD*

Suppose the query contains  $n$  terms:  $q_1, q_2, \dots, q_n$

1. Start with the query term that has the largest *maximum contribution*. In addition to building the scores up incrementally, collect the top  $k + 1$  scores into an accumulator.
2. If the sum of *maximum contributions* of the remaining query terms when added to the top  $k+1^{\text{th}}$  document (i.e., the highest one not in the top  $k$ ) does not improve its rank, then stop processing the inverted lists.
3. Otherwise, continue by starting with the term that has the next largest *maximum contribution* in the remaining inverted lists.

*Correctness:*

Claim: The *set* of top  $k$  documents produced by the modified algorithm (MOD) is identical to the set of top  $k$  documents produced by the original algorithm (ORIG).

Proof: If all query terms are processed then it is easy to see that MOD and ORIG produce the same set of top  $k$  documents. Therefore, assume that MOD skipped processing the inverted list for some subset of the query terms,  $Q' = \{q'_1, q'_2, \dots\}$ . Let,  $f_p(Q, k+1)$  denote the partial scores accumulated for the top  $k+1^{\text{th}}$  document, when MOD terminated. Then,

$$f_p(Q, k) \geq f_p(Q, k+1) + \sum_{q'_i \in Q'} \text{max\_contribution}(q'_i)$$

The summation on the right side of the above equation represents the maximum possible score that can be added to the top  $k+1^{\text{th}}$  document if all of the inverted lists in  $Q'$  had actually been processed. For any other document  $l$ , that is not already in the top  $k+1$  we must have,

$$f_p(Q, k+1) \geq f_p(Q, l) \quad \Rightarrow \quad f_p(Q, k) \geq f_p(Q, l) + \sum_{q'_i \in Q'} \text{max\_contribution}(q'_i)$$

Therefore, the top  $k$  documents produced by MOD and ORIG are identical sets.

*When will MOD be useless?*

MOD will be useless in cases where each inverted list contributes to improving the score of a small subset of the top  $k$  documents and does not impact the scoring of a majority. It will also fail if each of the terms is very important—has documents with high weights—so cannot be skipped.

*Modified Term-at-a-Time Algorithm: MOD'*

Suppose the query contains  $n$  terms:  $q_1, q_2, \dots, q_n$

1. Start with the query term that has the largest *maximum contribution*. In addition to building the scores up incrementally, collect the top  $k + 1$  scores into an accumulator.
2. Keep track of the smallest difference in scores in the top  $k+1$ , *min\_diff*.
3. Stop processing the inverted lists if both of the following conditions are met:
  - a. The sum of *maximum contributions* of the remaining query terms when added to the top  $k+1^{\text{th}}$  document does not improve its rank.
  - b. The sum of *maximum contributions* of the remaining query terms is smaller than *min\_diff*.
4. Otherwise, continue by starting with the term that has the next largest *maximum contribution* in the remaining inverted lists.

An obvious improvement would be to have the inverted lists sorted. This would allow us to process the inverted lists in decreasing order of document weights and stop processing as soon as processing more documents cannot yield any more changes to the current top  $k$  documents.

### **C. Compression (10 points)**

Consider the following inverted list for some term.

Docid=32, nTerms=3, pos=<16, 32, 128>, docid=35, nTerms=4, pos=<8, 24, 33, 48>

That inverted list could be represented as a sequence of 11 numbers as follows:

32, 3, 16, 32, 128, 35, 4, 8, 24, 33, 48

If each number is stored in a 32-bit integer, the sequence would occupy 352 bits (44 bytes). If the numbers were stored in the restricted variable-length encoding scheme in the lecture notes, using units of one byte (8 bits), how much space would it take?

Successively apply the three compression techniques discussed in class and in the texts to compress that list and show how big it is after each. So you'll have a result for delta encoding, then adding gamma codes, and finally adding delta codes.

Several different interpretations for the coding schemes (or what to apply them to) were used in the answers given. The most important 2 aspects of the answers were:

- 1) Your application was a reasonable interpretation of the code scheme as described by the lecture notes, and
- 2) Your application was consistently applied across the data

The example solution below is just one application of the codes discussed.

Given: (32, 3, <16,32,128>), (35,5,<8,24,33,48>)

**Restricted Variable-Length Encoding:**

Using one byte as the base unit, the RVL scheme is effectively the UTF-8 encoding. The value 128 requires 2 bytes to be encoded, whereas all other values in the list only require 1 byte. Therefore, the result requires 12 bytes = 96 bits of space, using 27.3% of the original space (96 / 352).

**Delta encoding:**

This transformation can't be uniformly applied on the list, as the components of the list are not sorted, nor can you expect a sorted condition to be an invariant over different operations during index construction.

However, you can perform the transformation on a field-by-field basis, noting that the docids across entries are sorted, as are the positions in each entry in the inverted list. Using this fact about our given list, the transform is:

(32, 3, <16,32,128>), (35,5,<8,24,33,48>) → (32, 3, <16,16,96>), (3,5,<8,16,9,15>)

Using no further compression, the resulting list still requires 44 bytes (352 bits), however if we apply the RVL scheme described above, the 128 value has been replaced by 96, its delta encoding value. 96 can be encoded in 1 byte using UTF-8, therefore we can store the resulting list in 11 bytes (88 bits) now, with a compression ratio of 25% (88/352).

**Gamma Code:**

We apply the gamma code to the result produced by delta encoding:

Given: (32, 3, <16,16,96>), (3,5,<8,16,9,15>)

Original	32	3	16	96	5	8	9	15
Gamma (Unary)	111110	10	11110	1111110	110	1110	1110	1110
(Binary)	00000	1	0000	100000	01	000	001	111

Size (bits)	11	3	9	13	5	7	7	7
-------------	----	---	---	----	---	---	---	---

We first construct a table of the mappings:

Using the provided table, the length of the inverted list after applying the gamma code is:  
 $11 + 3 + 9 + 9 + 13 + 3 + 5 + 7 + 9 + 7 + 7 = 83$  bits, a ratio of 23.5% (83/352) over 32-bit format.

**Delta Code:**

We now apply the delta code to the data after the delta encoding:

Given: (32, 3, <16,16,96>), (3,5,<8,16,9,15>)

Again, we construct a table of the mappings (note that the Gamma code of the length descriptor  $k_d$  is actually of  $k_d + 1$ ):

Original	32	3	16	96	5	8	9	15
$k_d$	5	1	4	6	2	3	3	3
$k_r$	0	1	0	32	1	0	1	7
$k_{dd}$	2	1	2	2	1	2	2	2
$k_{dr}$	2	0	1	3	1	0	0	0
Delta Code ( $\gamma$ of ( $k_d+1$ )) (binary)	110 10 00000	10 0 1	110 01 0000	110 11 100000	10 1 01	110 00 000	110 00 001	110 00 111
Size	10	4	9	11	5	8	8	8

Using this table, the length of the inverted list after applying the delta code is:  
 $10 + 4 + 9 + 9 + 11 + 4 + 5 + 8 + 9 + 8 + 8 = 85$  bits, producing a ratio of 24.1% (85/352).

**D. Clustering (10 points)**

Consider the algorithm for generating a single-link clustering of a document collection: start with singleton “clusters” and slowly lower the threshold, creating new links when the threshold falls below the similarity between documents. Suppose each document is represented as a node in a graph and that we insert an edge between nodes  $u$  and  $v$  only when the similarity between those nodes is the cause of a cluster merge. That is, if clusters  $C_1$  and  $C_2$  are about to be merged, there is some node  $u \in C_1$  and some node  $v \in C_2$  with maximal similarity causing the merge to happen: insert an edge between  $u$  and  $v$  (if there is a tie, choose randomly).

Prove that the resulting graph is a maximum spanning tree. That is, that it forms a tree, that it completely connects the graph, and that no other spanning tree will have a greater sum of edge weights (similarities).

Yes, *prove* means *prove*. It doesn't mean a hand-waving argument. (It should not be an overly complicated proof. The professor and TA came up with different techniques, so there are at least two valid solutions.)

Let  $T$  be the set of edges that we select. We first show that the set of edges form a tree.

Claim:  $T$  is a tree.

Proof: At each stage, all the clusters are trees. We use an inductive argument to prove this.

At the start, we have singleton clusters, each of which is trivially a tree. Assume that at some stage  $k$ , we have  $l$  clusters each of which is a tree. For the next stage, the algorithm cannot add any edge between nodes in the same cluster and therefore the edge added in stage  $k+1$ , cannot form any cycle. Thus, after stage  $k+1$ , we have  $l-1$  clusters each of which is a tree itself. Thus, the induction is complete.

Claim:  $T$  is a spanning tree. That is, it completely connects the graph.

Proof: If  $T$  does not connect the graph after the algorithm is completed, then there are two nodes  $u$  and  $v$  such that there is no path from  $u$  to  $v$  in the tree. But if no other edge connects them,  $u$  and  $v$  must eventually be directly connected by the algorithm (when the threshold falls to zero, in the extreme). That means that the algorithm is not finished.

Claim:  $T$  is a maximum spanning tree (MST).

Proof (by contradiction):

Assume that  $T$  is not a maximum spanning tree. Then there must be a maximum spanning tree  $T'$ , that differs from  $T$  and has a higher total weight. In particular, there must be at least one edge  $e(u,v)$  in  $T$  that is not in  $T'$ .

Removing  $e$ , will yield two connected components,  $C1$  and  $C2$ . Since  $T'$  is also a spanning tree, there must exist a different edge  $e'$  in  $T'$ , that connects  $C1$  and  $C2$ . Since  $e$  is in  $T$ ,  $e$  is the maximum similarity edge between  $C1$  and  $C2$ . Therefore, replacing  $e'$  with  $e$  in  $T'$  will either increase the weight of  $T'$  or keep it the same.

If  $T'$ 's weight increases as a result of the replacement, we have a contradiction to the assumption that  $T'$  is a maximum spanning tree. Otherwise, we can repeat the process for the remaining edges that are in  $T$  but not in  $T'$  until there are no such edges left or until we achieve a contradiction in the process. If all such edges are processed without any contradiction, we have shown that weight of  $T$  is in fact the same as  $T'$  which is a contradiction to the assumption that  $T$  is not a maximum spanning tree.