Assigning Document Identifiers to Enhance Compressibility of Fulltext Indices

(Extended Abstract)

F. Silvestri[1], R. Perego[1], S. Orlando[2]

[1] Istituto ISTI, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
[2] Dipartimento di Informatica, Università di Venezia, Mestre, Italy

Abstract. Index compression has been a major issue in the field of Information Retrieval Systems. In particular, due to the impressive figures involved with Web Search Engines (WSEs) the compression of the index is not an option anymore but it has become a must. The most important index compression methods are designed to work for *Inverted File* (IF) indexes. These methods are based on the assumption that the posting lists are stored as sequences of *d_gaps* (i.e. differences among successive document identifiers). The compression is thus carried out by using variable length encoding methods which represents smaller number using a smaller number of bits. In this paper, instead of focusing on finding a novel encoding method, we propose an algorithm which allows the assignment of identifiers to documents in a way that minimizes the average values of d_gaps. The simulations performed on a real dataset, i.e. the Google contest collection, show that our approach allows to obtain an IF index which is, depending on the d_gap encoding chosen, up to 23% smaller than the one built over randomly assigned document identifiers. Moreover, we will show, both analytically and empirically, that the complexity of our algorithm is linear in space and time.

1 Introduction

Web Search Engines (WSEs) processing steps are carried out by three distinct, and cooperating modules. The *Crawler* gathers Web documents and stores them into a huge repository. The *Indexer* abstracts the data contained within the document repository, and creates an *Index* granting fast access to document contents and structure. Finally, the *Searcher* module accepts user queries, searches the index for the documents that best match each query, and returns the *references* to these documents in an understandable form.

The main data structure used to represent the Index is the *Inverted File* [14] (IF). An IF, basically, consists of two distinct arrays containing: (a) the list of distinct terms contained in the collection, and (b) the *postings lists* that is an array of lists associated to each distinct term t and contains the *doc_ids* (which are numerical values) of all of the documents containing t. The success of this structure is due to some interesting properties that make it a suitable structure for WSE systems. In fact, an index adopting the IF data structure, will have a relatively small space occupancy and it could be efficiently used for resolving keyword–based queries [14]. The main operations involving posting lists basically require left to right scans of their elements. For this reason when a posting is accessed we can assume that we have already seen all the preceding postings. Furthermore, the posting lists are kept sorted by their doc_ids values. For these reasons, they are usually stored with a simple *difference coding* technique. For example, consider the posting list $((apple; 5)1, 4, 10, 20, 23)$ indicating that the term $apple$ occurs in 5 documents having doc_id $1, 4, 10, 20,$ and 23, respectively. The posting list can be rewritten as $((apple; 5)1, 3, 6, 10, 3)$, where the items of the list represent d_gaps, i.e. differences between successive doc_ids. Now, storing a d_gap posting list clearly requires the coding of smaller integer values, so it may benefits more from the adoption of variable length coding methods which encode smaller numbers with a smaller number of bits. From the discussion made so far, it is quite obvious that if we could devise an assignment of the doc_ids that globally minimizes the average value of the d_gaps present in all the postings lists then we may reduce the space needed to store the whole index structure. The problem, now, is: *how such an assignment can be devised*? The search space given by the set of all the permutations of n documents is exponential in n, and n is huge in the real case. So it is very likely that finding an optimal assignment of doc_ids will become soon intractable. In this paper we focused the attention in finding a scalable heuristic aimed at approximating an optimal reordering in a time proportional to the size n of the collection of documents. The proposed heuristic exploits a *text clustering* algorithm that reorders the collection on the basis of document similarity. The reordering is then used to assign close doc_ids to similar documents thus reducing d_gap values and enhancing the compressibility of the IF index representing the collection. At our best knowledge, the problem of reordering a collection of documents in order to obtain higher compression rates is addressed by only two other works [11, 1]. Unfortunately, the approaches proposed are expensive in both time and space and seem hardly applicable to huge collections. Since efficiency in time and space is obviously very important, we exploited a lightweight clustering algorithm to assign identifiers to documents. Our clustering algorithm resembles the *k-means* [2, 6]. The k-means algorithm works as follows. It initially chooses k documents as cluster representatives, and assigns the remaining $n - k$ documents to one of these clusters according to a given similarity metric. New centroids for the k clusters are then recomputed, and all the documents are reassigned according to their similarity with the new k centroids. The algorithm iterates until the position of the k centroids become stable. The main strength of this algorithm is the $O(n)$ space occupancy. On the other hand, computing the new centroids is expensive for large values of n, and the number of iterations required to converge may be high. We thus adopted a simplified version of the k-means algorithm requiring only k steps. At each step i the algorithm selects a document among those not yet assigned and uses it as centroid for the i-th cluster. Then it chooses among the unassigned documents the $\frac{n}{k} - 1$ ones most similar to the current centroid and assign them to the i-th cluster. The time and space complexity of our algorithm is the same of a single-pass *k-means*, but differently from classical clustering algorithms which produce clusters as sets of unordered items, our algorithm devises also an ordering among the members of the same cluster. Such ordering is exploited to assign consecutive doc_ids to consecutive documents belonging to the same cluster.

The rest of the paper is structured as follows. Section 2 discusses previous works regarding text clustering and the compression of IF indexes. Section 3 details our clustering algorithm and discusses its complexity. Section 4 describes the experiments conducted and presents the results obtained. Finally, Section 5 reports some conclusions along with a description of the research directions we plan to investigate in the near future.

2 Related Works

Most of the works regarding the compression of IF indexes deal with compression methods which we could call *passive*. With this term we intend emphasize the fact that they ”simply” exploit the locality present in the lists of postings to devise variable-length coding schema that minimizes the average number of bits (or bytes) required to store each doc_id [3, 5, 7, 9, 10, 8, 14]. Some of the most effective of such encoding methods are: *Gamma* [3], *Delta* [3], *Golomb* [5], and *Binary Interpolative* [8][1].

On the other hand, only two recent papers deal with *active* compression methods based on a preprocessing of the document collection aimed at reassigning the doc_ids in a way that enhances the compression rates achieved by the traditional, i.e. passive, encoding algorithms [11, 1].

In [11] the authors propose a doc_id reassignment algorithm that adopts a Travelling Salesman Problem heuristic. Each document of the collection is considered a vertex of a *similarity* graph $G = (V, E)$, where an edge is inserted between two vertexes d_i and d_j whenever the two documents associated to d_i and d_j share at least one term. Moreover, each edge $e \in E$ is weighted by the number of terms shared by the two documents. The TSP heuristic algorithm is then used to find a cycle in the graph having maximal weight and traversing each vertex exactly once. The suboptimal cycle found is finally used to reassign the doc_ids.

In [1] the authors propose an algorithm that permutes the document identifiers in order to enhance the locality of an IF index. A similarity graph $G = (V, E)$ is built as in the previous case but the *Cosine Measure* is used to weight the edges. Also this approach presents some drawbacks. First, as the one discussed in [11], it requires to store the whole graph G in main memory. Moreover, the cost of performing a split operation on the graph (at each level) is extremely high even if, to speed-up the algorithm, the authors propose two optimization heuristics that are used at each iteration to reduce the dimension of the subproblems. Finally, the algorithm starts from an already built IF (the bipartite graph terms-documents), and thus require a postprocessing step to reorder the IF and all the other linked files. The authors tested their strategy on an IF index built over the TREC-8 ad hoc track collection [13]. The results of several tests, conducted by varying the various parameters of the algorithm, were reported in the paper. The enhancement of the compression ratio obtained is significative, but the approach is high

[1] This method uses only implicitly the *d_gap* representation. In fact, it exploits the tendency of terms to appear relatively frequently in some parts of the collection, and infrequently in others.

demanding in terms of both time and space. The execution times reported in the paper regards tests conducted on a subcollection of only $32,000$ documents.

While these two papers address some relevant issues, the approaches proposed are probably unusable for large collections. Our proposal try to address this issue by exploiting a scalable clustering algorithm using space and time linear in the number of documents of the collection. This paper is a summary of the approach more deeply described in [12].

3 The Reordering Algorithm

Let $D = \{D_1, D_2, \ldots, D_N\}$ be a collection of N textual documents to which consecutive integer doc_ids $d = 1, \ldots, N$ are initially assigned. Moreover, let T be the number of distinct terms t_i, $i = 1, \ldots, T$ present in the documents.

Our clustering algorithm starts from a *transactional* representation of the documents: each document $D_i \in D$ is represented by the set $S_i \subseteq T$ of all the terms that the document contains. In the following we will denote the transactional document collection with $\widetilde{D} = \{S_1, S_2, \ldots, S_N\}$.

To evaluate the similarity among documents, we used the *Jaccard* measure [4].

Our *reordering* algorithm is outlined in Figure 1. It takes as input parameters the set $\widetilde{D}$, and the number k of clusters to create. In output it produces the ordered list of all the members of the k clusters. This list univocally defines π, a reordering of $\widetilde{D}$ minimizing the average value of the d_gaps.

Algorithm 1 The reordering algorithm.

```
 1: Input:
      – The set $\widetilde{D}$.
      – The number $k$ of clusters to create.
 2: Output:
      – $k$ ordered clusters representing a reordering $\pi$ of $\widetilde{D}$.
 3: for $i = 1, \ldots, k$ do
 4:   $\textit{current\_center} = \textit{longest\_member}\left(\widetilde{D}\right)$
 5:   $\widetilde{D} = \widetilde{D} \setminus \textit{current\_center}$
 6:   for all $S_j \in \widetilde{\mathrm{D}}$ do
 7:     $\textit{sim}[j] = \textit{compute\_jaccard}(\textit{current\_center}, S_j)$
 8:   end for
 9:   $M = \textit{select\_members}(\textit{sim})$
10:   $c_i = c_i \cup M$
11:   $\widetilde{D} = \widetilde{D} \setminus M$
12:   $c_i = c_i \cup \textit{current\_center}$
13:   $\textit{dump}(c_i)$
14: end for
```

The algorithm performs k scans of $\widetilde{D}$. At each scan i, it chooses the longest document not yet assigned to a cluster as current center of cluster c_i, and computes the distances between it and each of the remaining unassigned documents. Once all the similarities have been computed, the algorithm selects the $\left(\left|\widetilde{D}\right|/k\right) - 1$ documents most similar to the current center by means of the procedure *select_members*.

The algorithm returns a ordered list of documents. The corresponding reordering π assigns doc_id i to the i-th document of the returned list.

In [12] we computed the time and space complexity of this algorithm. The total time spent by the algorithm is given by

$$T(|\widetilde{D}|, k) = O\left(|\widetilde{D}|k\right)$$

The space occupied by the reordering algorithm is given by the following equation:

$$S(|\widetilde{D}|, k) = O\left(|\widetilde{D}|\right)$$

4 Experimental results

To assess the performances of our algorithm we tested it on a real collection of Web documents, the publicly available Google Programming Contest collection[2]. This collection consists of about $900,000$ documents, and occupies about 2GBytes of space on disk. In this section we present some preliminary results obtained running our algorithm on a Linux PC using a 2GHz Intel Xeon processor, 1GB of main memory, and an Ultra-ATA 60GB disk.

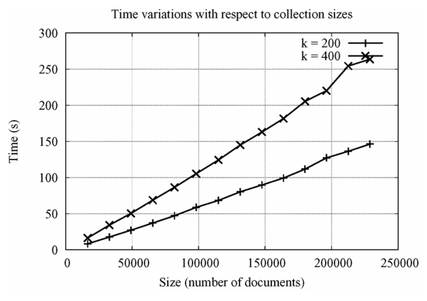
The tests we performed were aimed to evaluate efficiency and effectiveness of our solution under several configurations. In particular we varied the number of documents to reorder and the number of clusters generated.

In Figure 1.(a) the algorithm execution time is plotted as a function of the size of the collection. The two curves reported correspond to different values of k. In both cases the curves have a linear behavior as we argued from the theoretical analysis outlined in the previous section.

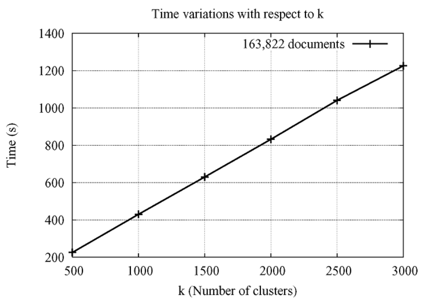
From the above analysis results that the time spent in reordering also depends from k. In order to evaluate the completion times as a function of k, we performed a test on a subset of the collection composed by about $160,000$ documents. Figure 1.(b) shows the results of this test that confirms our analytical estimate.

In Figure 2 the space used by the reordering algorithm is plotted as a function of the size of the collection. As it can be seen, the space needed by the algorithm is linear in the number of documents being reordered, and can be estimated to be approximately equal to $1\text{KByte} \cdot \left|\widetilde{D}\right|$. Since memory occupancy is dominated by the space used to store the collection in main memory it does not depend from the number of clusters generated.

[2] http://www.google.com/programming_contest



(a)



(b)

Fig. 1. Execution times as a function of: (a) the size of the collection; (b) the number of clusters.

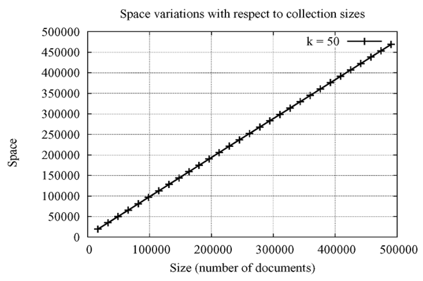


Fig. 2. Space occupied by the reordering algorithm by varying the size of the collection.

The main goal of our reordering algorithm is producing an assignment of identifiers that increases the number of small d-gaps.

To verify this deduction we computed the average number of bits needed to store each doc_ids for randomly assigned document identifiers and reordered ones. We also varied various parameters such as the size of the collection, the number of clusters k, and the encoding method used. Tables 1 and 2 reports the results obtained.

From both the tables we can observe a significative reduction in the average number of bits-per-posting value for all of the encoding schemes used. For the whole collection (see Table 1), the maximum compression rate enhancement was achieved by adopting the Binary Interpolative scheme. In this case our algorithm allowed to reduce the size of the compressed index of an additional 14%. Also from the data reported in Table 2, we can observe that our algorithm produces an high improvement in the compression rates. In the case of Gamma encoding, we improve the average number of bits-per-posting from 9.1 to 6.9. This reduction corresponds to an IF index 23.2% smaller than the one built over randomly assigned identifiers. Moreover, from the results reported in both the tables we can deduce that acceptably good reorderings are obtained also with relatively small values of k. For example let us to consider the bits-per-posting values

k	Gamma	Delta	Golomb	Interpolative
random	9.0157	8.5882	7.1382	6.9644
100	8.2478	7.7207	6.4676	6.1117
500	8.1146	7.6083	6.4670	6.0449
2000	8.0162	7.5166	6.4666	6.0050
8000	7.9293	7.4335	6.4664	**5.9874**
9000	**7.9225**	**7.4268**	**6.4663**	5.9876

Table 1. Average number of bits-per-posting for different values of k and different encoding methods. *random* == no assignment performed.

k	Gamma	Delta	Golomb	Interpolative
random	9.0860	8.4330	6.4753	6.5644
5	7.5552	7.1661	6.3807	5.7129
100	7.3106	6.9410	6.3505	5.5563
500	7.1106	6.7510	6.3531	5.4535
2000	6.9981	6.6389	**6.3523**	**5.4248**
3500	**6.9751**	**6.6102**	6.3524	5.4290

Table 2. Average number of bits-per-posting for different values of k and different encoding methods. The results reported refer to tests conducted with a subset of the Google collection composed of about $160,000$ web documents. *random* == no assignment performed.

shown in table 2 in correspondence of the Binary Interpolative Coding method. For values of k equal to 500 and 2000 the bits-per-posting value resulted equal to 5.4535 and 5.4248, respectively. This means that we loose only about 0.5% in compression rate when we generate 500 instead of 2000 clusters. However, by looking at the plot reported in Figure 1 we can see that reordering the collection with $k = 500$ takes about one fourth of the time required when $k = 2000$.

5 Conclusions

In this paper we presented an efficient algorithm for computing a reordering of a collection of textual documents that effectively enhances the compressibility of the IF index built over the reordered collection. The reordering algorithm exploits a k-means-like clustering algorithm which compute an effective reordering with linear space, and time complexities. The experimental evaluation conducted with a real world test collection, resulted in improvements up to $\sim 23\%$ in the compression rate achieved.

The algorithm depends on a parameter k which indicates the number of clusters produced. The quality of the reordering obtained is however only marginally sensible to the value of k. Large values of k results in the average in slightly better compression rates. On the other hand, smaller values of k result in a significantly lower completion time.

In the near future we plan to extend our work in several directions. First, we will compare our solution with the algorithm proposed in [1]. Even if the Blelloch algorithm should produce slightly better reorderings than ours, we think that our approach may reach comparable compression rates by using a significantly lower amount of computational resources. Moreover, we plan to perform a more deep and accurate experimental evaluation on larger collections of Web documents. Finally, since collections coming from real Web repositories typically contain several millions of documents, we plan to design and evaluate an on-line version of our algorithm where the reordering is applied to fixed-size blocks of documents.

References

1. Dan Bladford and Guy Blelloch. Index compression through document reordering. In IEEE, editor, *Proceedings of the DATA COMPRESSION CONFERENCE (DCC'02)*. IEEE, 2002.
2. Soumen Chakrabarti. *Mining the Web - Discovering Knowledge from Hypertext Data*. Morgan Kaufmann Publishers, San Francisco, 2003.
3. P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, February 1975.
4. W. B. Frakes and Editors R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*, chapter Clustering Algorithms (E. Rasmussen). Prentice Hall, Englewood Cliffs, NJ, 1992.
5. S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.
6. A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentiche Hall, 1988.
7. A. Moffat and T. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.
8. Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.
9. Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. of the 21st Intl. Conf. on Research and Development in Information Retrieval*, pages 290–297, October 1998.
10. Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted index for fast query evaluation. In *SIGIR02*, 2002.
11. Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shann, and Chung-Ping Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1):117–131, January 2003.
12. Fabrizio Silvestri, Raffaele Perego, and Salvatore Orlando. Assigning document identifiers to enhance compressibility of web search indexes. In *Proceedings of the Symposium on Applied Computing (SAC) - Special Track on Data Mining (DM)*, Nicosia, Cyprus, March 14–17 2004. ACM.
13. E. Voorhees and E. D. K. Harman. Overview of the eighth text retrieval conference (trec-8). In *Proceedings of the Eighth Text REtrieval Conference (TREC-8)*, pages 1– 24, 1999.
14. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes – Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, second edition edition, 1999.