Optimize Document Identifier Assignment for Inverted Index Compression

Chong CHEN\textsuperscript{1,†}, Jing HE\textsuperscript{2}, Dongdong SHAN\textsuperscript{2}, Hongfei YAN\textsuperscript{2}

\textsuperscript{1}Department of Information Management, Beijing Normal University, Beijing, 100875, China
\textsuperscript{2}Department of Computer Science, Peking University, Beijing, 100871, China

Abstract

Document identifier assignment is a technique for inverted file index compression, by reducing d-gap value of posting lists. It was approached by either TSP or clustering methods in existing study. However, there is no proper formulation for this problem and the existing approaches has no theory guarantee to be good approximations. In this paper, we first formulate document identifier assignment problem as an optimization problem, and then propose a new method to solve it approximately. Our method first clusters the documents by URL information and then rearranges the documents and clusters with benefit function, which is derived by minimizing posting space directly. TSP method can be considered as one simple case of our method. The experiments show that it achieves a good trade-off between efficiency and effectiveness.

Keywords: Document Identifier; Cluster; Inverted Index Compression; Optimization

1. Introduction

Inverted file index is an important component for IR system. When the dataset becomes larger, index efficiency is a challenge. Compression is one technique to improve inverted file index efficiency for two reasons: 1) it reduces the disk space for index; 2) it reduces loading time in query evaluation. Inverted index is composed by a collection of posting lists, each of which stores the identifiers of documents containing one term. The documents in a posting list are sorted by either document identifiers or impact. For the first type, in a posting list, the first posting stores original document identifier and the subsequent postings store d-gaps between two neighbor identifiers\textsuperscript{[10]}.

Inverted file index compression is to compress the collection of d-gap values. There are two categories of compression algorithms: bitwise\textsuperscript{[10]} and bytewise\textsuperscript{[1]}. There are some research on how to change the d-gap distribution to improve compression ratio. Almost all compression algorithms have the property that it takes smaller space for smaller value. Therefore, the compression can be improved by decreasing d-gap values. One effective approach for this task is to reassign document identifier. For example, for documents $A, B, C, D$ and terms $t_1, t_2, t_3$, the document-term matrix is shown in Table 1. In the original document assignment $d_1 = A, d_2 = B, d_3 = C, d_4 = D$, the posting lists are $l_1 = (1, 3), l_2 = (2, 4), l_3 = (1, 3)$, for term $t_3$.  

\textsuperscript{†} Corresponding author.

Email addresses: chenchong@bnu.edu.cn (Chong CHEN)

1553-9105/ Copyright © 2010 Binary Information Press
February, 2010
After converting original document identifier to \( d \)-gap, the posting lists become \( l_1 = (1, 2), \ l_2 = (2, 2), \ l_3 = (1, 2) \). However, if we reassign the document identifiers to be \( d_1 = A, d_2 = C, d_3 = B, d_4 = D \), the \( d \)-gap posting lists become \( l_1 = (1, 1), \ l_2 = (3, 1), \ l_3 = (1, 1) \). It obvious that the overall values in the posting lists become smaller. In this example, we assign close identifiers for document pairs \((A, C)\) and \((B, D)\), whose content are very similar. More generally, there are two similar documents \( d \) and \( d_0 \), which share a large number of words. If it assigns document identifiers \( i \) and \( i + 1 \) to them separately, there would be a \( d \)-gap 1 for each common word. If assigned identifiers far away, \( d \)-gap becomes larger. The property is named as clustering property\([9]\), and document identifier assignment is motivated by this intuition.

<table>
<thead>
<tr>
<th>Table 1 Simple Example for Document Identifier Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
</tr>
<tr>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( C )</td>
</tr>
<tr>
<td>( D )</td>
</tr>
</tbody>
</table>

The document identifier assignment problem is either approximated with traveling salesman problem (TSP) or clustering problem. Both approaches are inspired by the clustering property. However, there is no formal method to determine a better approximation. In this paper, we formulate it as an optimization problem, and derive an approximation algorithm for it. It’s a hybrid algorithm, combining clustering and benefit function based solutions. TSP method can be considered as one special simple case of the benefit based solution. The result shows that it is a flexible trade-off between efficiency and effectiveness.

**Related work**

Shieh\([7]\) approximated document identifier assignment problem as TSP. However, the exact solution of TSP is not necessary the optimal solution to document identifier assignment problem, because it ignores the documents whose identifiers differ more than 1. Shieh approximates TSP problem by greedy-NN algorithm, but it still has efficiency problem. To overcome the weakness, Blanco and Barreiro\([4,3]\) proposed two variance methods for TSP problem. The first method is to just store a compressed document similarity graph by Singular Value Decomposition(SVD) approach. The second method(c-block) splits the document collection into blocks, and then runs greedy-NN algorithm both intra-block and inter-blocks. These two method reduced computation time a lot, but they are still not very scalable.

Another effort is to solve this problem by clustering approach. Blelloch and Blandford\([5]\) used clustering techniques firstly. Like Shieh’s method, it’s also not scalable because it needs to store a complete document similarity graph. Some work followed to make it more efficient. \([9]\) proposed a series of one-pass clustering techniques and it doesn’t require a pre-built index. The most efficient method in this category is URL-sorting method\([8]\). In this method, it sorts the documents according to their URLs and assigns document identifiers according the URL order. This method performs well and scalable. However, there is no guarantee that good clustering result necessarily leads good compression result.

The only formulation work on this problem is by Blanco and Barreiro\([2]\), formulating it as Pattern Sequence Problem(PSP). However, such formulation cannot present the real required space, because
compression algorithm usually compresses a value in much less number of bits than the actual value. We should reformulate it with a more reasonable objective function.

2. Optimization Problem

For a document collection, the inverted file index is composed by a collection of posting lists, each one of which is a list of document identifiers in increasing order. The document-term information is always presented by a matrix $M$. Each row of it presents a document and the $i$-th row is for document with identifier $i$. Each column presents a term. $M_{ij}$ is a binary value, indicating whether document $i$ contains term $j$. One example is shown in Table 1. The document identifier problem is to find of row permutation $\pi$ of the matrix $M$, so that the inverted file index space required for permuted matrix $M_\pi$ is minimal. The optimization problem can be formulated as below:

$$
\pi_0 = \arg \min_{\pi} \sum_{j=0}^{\left| T \right|} s(\pi(d_{j})) + \sum_{k=2}^{I} s(\pi(d_{jk}) - \pi(d_{jk-1}))
$$

Where $|T|$ is total number of unique terms in the collection, $f_j$ is document frequency for term $t_j$, and $(\pi(d_1), \ldots, \pi(d_{|T|}))$ is the identifiers of documents who contain term $t_j$ after permutation. $s$ is a storage space function presents how many bits required to a integer value. Here we name $\pi(d_k) - \pi(d_{k-1})$ to be distance between neighbor document in posting list. In [2], it has not introduced storage space function $s$ in its formulation, and default uses unary encoding for their presentation. However, actually no inverted file index is compressed with unary encoding because it spends too much space. Our formulation, however, can be more generally adapted to various of compression algorithms. With this formulation, we have the principle to select the weight function and approximate algorithm.

3. Approximate Algorithm

In this section, we propose an algorithm to solve the optimization problem approximately. First, we propose the framework of URL tree resort algorithm. And then we introduce two important components: document collection ordering algorithm and document subsets ordering algorithm. We also derive the weight function based on posting storage space.

3.1. URL Tree Resort Algorithm

We propose URL tree resort algorithm: it uses URL information to build a URL tree as first step for its efficiency and employs content similarity based method to boost its effectiveness. The pseudo code for this algorithm is presented in Algorithm 1.

This algorithm firstly builds a document tree(buildTree) by URL information. The root of the tree is a pseudo node and the second-level nodes are all sites. The other inner nodes are directories and the leaf nodes are pages. The document tree keeps only clustering information but not ordering information, because clustering by site/directory is effective. The ordering information is achieved by function orderDocument and orderSubset. While pre-order visiting the tree, it orders documents for small document collection, whose size is less than threshold $t$, and then orders subsets for large node. Because it is pre-order visiting, the documents in subsets must be in order before it orders these subsets. For example, one
document tree is presented in Figure 1. The document tree is composed by three sites: \(A\), \(B\) and \(C\). \(A\) has three directories: \(A_1, A_2, A_3\), \(B\) has two directories \(B_1, B_2\) and \(C\) has no directory. It assumes that all the directories has less documents than threshold \(t\). After building this tree as first step, for site \(A\), it orders the documents in directories \(A_1, A_2\) and \(A_3\) by \text{orderDocument} function separately, and then orders subsets \(A_1, A_2, A_3\) in sites \(A\) by \text{orderSubset} function. It works similarly for sites \(B\) and \(C\). After all documents in site \(A, B, C\) are ordered, it orders \(A, B, C\) by \text{orderSubset} function.

One approach to order the document collection and subset is to employ TSP method multiple times. This approach is similar to c-blocks algorithm[4]. There are two differences between our algorithm and c-block:

- c-block method divides fixed-size blocks, but our numbers of documents in blocks are not fixed.
- c-block method has a set of flat blocks, but our clusters are organized hierarchically.

A drawback of TSP method is that it only considers the documents whose identifier differs by 1. However, in our problem formulation, the space required is determined by neighbor documents of posting lists, and the neighbor documents in posting list are not necessary have identifier difference 1. Intuitively, it can improve compression performance by considering more documents whose identifiers differ larger than 1. One problem is that the weight between two documents is affected not only by the content of two documents, but also the document identifier difference and content of documents between them. The algorithm and the weight function is discussed in section 3.2, also based on benefit function.

The subset ordering method in c-block is simply by ordering samples selected from subsets. It has no guarantee that a good order of samples leads good order of subsets, so we propose another ordering algorithm for subsets in section 3.3.

### 3.2. Order Document Collection

We propose a greedy algorithm to approximate document identifier assignment problem, and this algorithm is the implementation of \text{orderDocument} function in Algorithm 1.

This algorithm selects a document randomly from the collection as the first document, assigning identifier 1. With documents \(d_1, \ldots, d_h\), which have been assigned identifiers \(1, \ldots, i\), it assigns identifier
$i + 1$ to the document which leads maximal storage space benefit. The pseudo code for this algorithm is presented in Algorithm 2.

```
Input: unassigned identifier document collection $D$
Output: ordered document queue $q = (d_1, \ldots, d_N)$
begin
    remove document $d_0$ randomly from collection $D$
    $q$.append($d_0$)
    while $D$ is not empty do
        $d_0 = \arg\max_d \text{weightFunction}(d, q, D)$
        remove $d_0$ from collection $D$
        $q$.append($d_0$)
    end

Algorithm 2: Order Document Collection
```

Weight function is a very important component for this algorithm and other existing methods. There are various weight functions used, most of which are based on content similarity of two documents. In [9], it used Jaccard similarity coefficient to measure the similarity. Another common used similarity measure is cosine coefficient. However, they are just “borrowed” from other applications. It’s lack of validation that they are effective for document identifier assignment problem.

We derive a new weight function from the optimization problem defined in section 3. We propose a benefit based weight function. The main idea of this function is to measure how much “benefit” we can achieve by assigning a document identifier. For example, if we have assigned document identifier for $d_1, \ldots, d_i$, and $D$ is the document set which are not assigned identifiers, we will calculate how much benefit we can achieve by assigning identifier $i + 1$ to document $d \in D$.

Firstly, let’s consider last assigned document $d_i$ and document to be assigned $d$. There are three types of terms in $d_i$ or $d$: the common terms($t \in d_i \cap d$), the terms belonging to $d_i$ only($t \in d_i \setminus d$) and those belonging to $d$ only($t \in d \setminus d_i$). If assigned identifier $i + 1$ to $d$, for a common term $t$, the storage space in the posting list after $d_i$ is composed by two parts: one posting for storing d-gap between $d_i$ and $d$, and the number of postings for storing the d-gaps between the unassigned documents containing term $t$. The first part is simply $s(1)$, and the second part is $(df(t, D) - 1) \cdot s(df(t, D) - 1) \cdot \mid D \mid - 1$, where $s(n)$ is storage space for value $n$, $df(t, D)$ is document frequency for $t$ in collection $D$ and $s(df, dn)$ is a function to calculate the space expectation for term $t$ with document frequency $df$ and total document number $dn$. If assigning document identifier randomly, the d-gap distribution should be a geometric distribution, whose mean is $dn \cdot df + 1$.

Therefore the posting space expectation for a term $t$ should be:

$$
\bar{s}(df(t), dn) = \sum_{k=1}^{dn-df(t)+1} Pr(dgap = k) \cdot s(k)
$$

$$
= \sum_{k=1}^{dn-df(t)+1} (1 - p(t))^{k-1} \cdot p(t) \cdot s(k)
$$

$$
p(t) = \frac{df(t) + 1}{dn}
$$

If not assigning identifier $i + 1$ to document $d$, for a common term $t$, the space expectation for a posting list after $d_i$ is $s(df(t, D), \mid D \mid)$. In summary, the benefit from a common term is presented in Equations 2.
Similarly, we can also calculate the benefit from other two types of terms, but we ignore them because it is much smaller.

\[ b(t, D) = s_{u(t, D)} - s_{a(t, D)} \]

\[ s_{u(t, D)} = df(t, D) \cdot \overline{S}(df(t, D), |D|) \]

\[ s_{a(t, D)} = S(1) + (df(t, D) - 1) \cdot \overline{S}(df(t, D) - 1, |D| - 1) \]  

(2)

So the overall benefit from \( d \) and \( d \) is the sum of all common words of \( d \) and \( d \):

\[ b(d, d, D) = \sum_{t \in d \cap d} b(t, D) \]

More generally, there is still benefit from \( d_{i-k} \) and \( d \) by assigning identifier \( i + 1 \) to document \( d \). Similarly, the benefit can be summed up by all common words of them, but the benefit function for each common word is different from that of \( d_i \) and \( d \). First, it should consider the distance between the documents: the longer distance leads less benefit because it costs more storage space for longer distance; Second, it should consider the documents \( \{d_{i-k}, \ldots, d\} \), which are between \( d_{i-k} \) and \( d \): if any of these documents contain the common word \( t \), the benefit is “blocked” by the document, because there is no direct d-gap between \( d_{i-k} \) and \( d \). For a list of assigned identifier documents \( q = (d_1, \ldots, d) \), the benefit of assigning identifier \( i + 1 \) to document \( d \) is defined the sum of benefits between \( d \) and all assigned identifiers documents.

\[ b(d, q, D) = \sum_{t \in d \cap d} b(t, D) \]

\[ b(d_j, d, q, D) = \sum_{t \in d \cap d} b(t, j-i, D) \]

\[ b(t, k, D) = s_{u(t, k, D)} - s_{a(t, k, D)} \]

\[ s_{u(t, k, D)} = \overline{S}(df(t, D), |D|, k) + (df(t, D) - 1) \cdot \overline{S}(df(t, D) - 1, |D| - 1) \]

\[ s_{a(t, k, D)} = S(k + 1) + (df(t, D) - 1) \cdot \overline{S}(df(t, D) - 1, |D| - 1) \]  

(3)

This benefit function is more powerful and effective than existing weight function for two reasons: 1) it is derived directly from the storage space, but not just employing other measure for content similarity arbitrarily; 2) it considers document identifier difference, which is required in our URL resort algorithm.

Now we are going to specifically define the storage space function \( s \). The upper bound for lossless compression algorithm for a positive integer \( n \) is \( \log_2(n) \). Currently, there are many state-of-art algorithms approaching this upper bound, so we approximate storage function as \( \log_2 \).

In this formulation, it calculates the benefit by summing up benefits from all assigned documents. If the document collection is large, the document far away affects little to the overall benefit, because more common words are blocked and benefit decreases as distance increases. One alternative approach is to set a threshold \( \text{dist} \), and only the documents whose identifier distance is smaller than this threshold are considered. When the threshold dist is to set the threshold to be 1, it is same as the greedyNN algorithm for TSP. So we can see that the existing TSP method is a simple special case of our approach.

3.3. Order Document Subsets

As presented in Algorithm 1, the document subsets are ordered after the documents in the subsets are ordered. The weight between a subset \( D \) and ordered subsets \( q \) can be evaluated by the benefit to put the
subset next the ordered subsets, presented in Equation 4, where \( D \) is all documents in the unsigned identifier subsets and benefit function \( b(d,q,D) \) from document \( d \), document list \( q \) and unsigned document collection \( D \) is defined in Equation 3.

\[
b(q, D, D') = \sum_{d_i \subseteq D} b(d_i, q + D_{i,j-1}, D')
\]  

4. Experiment and Result Analysis

4.1. Document Identifier Assignment Algorithms

We perform experiments on WT2g dataset, containing 250,000 documents. We use a Pentium IV 4.0GHz, with 2GB of RAM and Redhat Linux system. We consider Oracle, Gamma and Simple as compression methods. Oracle is an optimal encoding ratio. Gamma and Simple present the results for bitwise and byte-wise algorithms respectively. We use URL sorting method as our baseline. The result is presented in Table 2. The best performance of document assignment algorithm for each compression algorithm is in bold. This result shows that our method can achieve much better result using bitwise compression, but not so with byte-wise algorithm.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Oracle</th>
<th>Gamma</th>
<th>Simple9</th>
<th>AssignTime(s)</th>
<th>TotalTime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>3.91</td>
<td>9.22</td>
<td>9.61</td>
<td>-</td>
<td>587.3</td>
</tr>
<tr>
<td>URL Sorting</td>
<td>2.49</td>
<td>6.85</td>
<td>8.15</td>
<td>18.0</td>
<td>605.3</td>
</tr>
<tr>
<td>URL Tree Resort</td>
<td>2.28</td>
<td>6.58</td>
<td>8.08</td>
<td>165.6</td>
<td>752.9</td>
</tr>
<tr>
<td>100K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>4.14</td>
<td>9.39</td>
<td>10.10</td>
<td>-</td>
<td>1188.9</td>
</tr>
<tr>
<td>URL Sorting</td>
<td>2.61</td>
<td>7.10</td>
<td>8.50</td>
<td>29.2</td>
<td>1218.1</td>
</tr>
<tr>
<td>URL Tree Resort</td>
<td>2.37</td>
<td>6.75</td>
<td>8.41</td>
<td>347.0</td>
<td>1335.9</td>
</tr>
<tr>
<td>150K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>4.19</td>
<td>9.78</td>
<td>10.21</td>
<td>-</td>
<td>1807.8</td>
</tr>
<tr>
<td>URL Sorting</td>
<td>2.66</td>
<td>7.17</td>
<td>8.52</td>
<td>44.7</td>
<td>1852.5</td>
</tr>
<tr>
<td>URL Tree Resort</td>
<td>2.48</td>
<td>6.88</td>
<td>8.44</td>
<td>498.8</td>
<td>2306.6</td>
</tr>
<tr>
<td>200K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>4.22</td>
<td>9.82</td>
<td>10.17</td>
<td>-</td>
<td>2425.3</td>
</tr>
<tr>
<td>URL Sorting</td>
<td>2.64</td>
<td>7.17</td>
<td>8.45</td>
<td>61.9</td>
<td>2487.2</td>
</tr>
<tr>
<td>URL Tree Resort</td>
<td>2.42</td>
<td>6.87</td>
<td>8.42</td>
<td>702.8</td>
<td>3128.1</td>
</tr>
<tr>
<td>250K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>4.32</td>
<td>10.02</td>
<td>10.33</td>
<td>-</td>
<td>3223.6</td>
</tr>
<tr>
<td>URL Sorting</td>
<td>2.65</td>
<td>7.18</td>
<td>8.58</td>
<td>82.5</td>
<td>3306.1</td>
</tr>
<tr>
<td>URL Tree Resort</td>
<td>2.42</td>
<td>6.89</td>
<td>8.44</td>
<td>941.3</td>
<td>4164.9</td>
</tr>
</tbody>
</table>

For computation complexity, URL tree resort algorithm takes longer time than URL sorting algorithm because it needs to analyze the content of Web page. However, as we stated, the complexity of the algorithm is also \( O(n\log n) \), which is same as URL sorting algorithm. Furthermore, we argue that: 1) the compression ratio affects online retrieval performance, so it’s much important than offline indexing performance; 2) the identifier assignment is not the bottleneck of indexing. In the 7th column of Table 2,
we show the total time of indexing plus document identifier assignment. Compared to it, URL sorting and tree resorting algorithm do not increase a lot of indexing cost.

4.2. Weight Functions

The existing work has no empirical comparison between different weight functions. We compare the three types of weight functions: Jaccard, cosine and benefit based weight function. The first experiment is to use URL sort algorithm, but with different weight functions. The results are reported in table 3. The experiment shows that benefit based function > Jaccard > Cosine.

5. Conclusion

We formulate document identifier assignment problem and present URL tree resort algorithm to solve it approximately. The experiment results show that URL tree resort algorithm is a good and flexible trade-off between effectiveness and efficiency. Benefit based weight function outperforms the other weight functions.

Acknowledgement

This work is partially supported by NSFC under the grant number 70903008 and 60933004, CNGI grant number 2008-122 and 863 Program No. 2009AA01Z143.

References