行政院國家科學委員會專題研究計畫　期中進度報告

具有動態資源管理之高效能叢集式資訊檢索系統設計(1/3)

計畫類別：個別型計畫
計畫編號：NSC92-2213-E-009-065-
執行期間：92 年 08 月 01 日至 93 年 07 月 31 日
執行單位：國立交通大學資訊工程學系

計畫主持人：單智君
共同主持人：鍾崇斌
計畫參與人員：鄭哲聖

報告類型：精簡報告
報告附件：出席國際會議研究心得報告及發表論文
處理方式：本計畫可公開查詢

中　華　民　國 93 年 5 月 14 日

中文摘要

網際網路的迅速成長，為資訊檢索系統的設計，帶來了高效能與低成本的挑戰。為了服務網路上每秒成千上萬個的使用者需求，資訊檢索系統需要索引結構來加速資料的搜尋。這個報告針對目前最熱門的索引結構—轉置檔案，提出一個新的壓縮機制。在轉置檔案中，每一個字彙都有一個相對應的文件編號串列(稱為轉置串列)來指示那一個文件包含這個字彙。我們觀察到在一個轉置串列中，文件編號的分佈具有群聚性。一個好的轉置檔案壓縮法不但必須能夠妥善地利用這種文件編號的群聚性來達到有效率的壓縮，更要能夠快速的解壓縮。在這個報告中，我們提出了一個新的壓縮法，它以內插編碼法為基礎，並透過 γ 編碼法與 Golomb 編碼法等 d-gap 壓縮法來加快編碼/解碼速度。這個新的壓縮法，我們稱為『單一順序(unique-order)的內插編碼法』。與內插編碼法比較，我們所提的方法在編碼/解碼的過程中並不需要利用 stack，因此編碼/解碼的速度都較內插法來得快。並且這個新的壓縮方法也可以妥善地利用文件編號的群聚性，因此壓縮效率也相當地好。與其他已知的壓縮法比較，我們所提的壓縮法提供了快速的解壓縮與良好的壓縮效能。我們也觀察到這個方法可以很容易地支援 self-indexing strategy，大幅加快查詢的處理速度。我們相信此一新的壓縮方法可以應用於高效能與低成本的資訊檢索系統設計。

關鍵字: 轉置索引壓縮, 轉置檔案, prefix-free 編碼法, 內插編碼法, 快速解碼

英文摘要

The rapid growth on Internet brings challenges on not only high performance but also low cost for information retrieval system (IRS) design. To service thousands of requests arriving for one second, IRSes require an indexing structure so that the desired data can be located quickly. This report presents a size reduction method for the inverted file, the most suitable indexing structure for an information retrieval system (IRS). We notice that in an inverted file the document identifiers for a given word are usually clustered. While this clustering property can be used in reducing the size of the inverted file, good compression scheme as well as easy decompression must both be available. In this report, we present a method that can facilitate coding and decoding processes for interpolative coding using only simple and high-speed models such as γ coding and Golomb coding in *d*-gap technique. We call this method the unique-order interpolative coding. It can calculate the lower and upper bounds of every document identifier for a binary code without using a stack, hence the decompression time can be greatly reduced. Moreover, it also can exploit document identifiers clustering and compress the inverted file efficiently. Compared with the other well-known compression methods, our method does provide fast decoding speed and excellent compression result. This method can also be used to support the self-indexing strategy. Therefore our research work in this paper provides a feasible way to build a fast and space-economical IRS.

Keywords: inverted index compression, inverted file, prefix-free coding, interpolative coding, fast decoding

報告內容： (Presented at Proceedings of ITCC 2004 International Conference on Information Technology: Coding and Computing, Vol. 2, pp. 229-235, Apr. 2004, Las Vegas, Nevada, USA.)

A Unique-Order Interpolative Code for Fast Querying and Space-Efficient Indexing in Information Retrieval Systems

Cher-Sheng Cheng, Jean Jyh-Jiun Shann, and Chung-Ping Chung
Department of Computer Science and Information Engineering,
National Chiao-Tung University, Hsinchu 30050, Taiwan.
{jerry, jjshann, cpchung}@csie.nctu.edu.tw

Abstract

The word positions for any given word in the whole collection are arranged in clusters. If we can use the method that can take advantage of clustering, excellent results can be achieved in compression of inverted file. However, the mechanisms of decoding in all the well-known compression methods that can exploit clustering are more complex, which reduce the ability of searching performance in information retrieval system (IRS) at some degree. In this paper, we proposed a new method that can facilitate coding and decoding of interpolative code by using the simply applied and high-speed models such as γ code and Golomb code in d-gap technique. This new method can exploit clustering well, and the experimental results confirm that our method can provide fast decoding speed and excellent compression efficiency.

1. Introduction

An inverted file contains, for each distinct term t in the collection, an inverted list of the form $IL_t = \langle f_t, id_1, id_2, \ldots, id_{ft} \rangle$, where frequency f_t is the total number of documents in which t appears, and id_i is the identifier of the document that contains t. To process a query, the information retrieval system (IRS) retrieve the inverted lists of the terms appearing in the query, and then perform some set operations, such as intersection (∩) and union (U), on the inverted lists to obtain the answer list [1][2].

A popular compression technique is to sort the document identifiers of each inverted list in increasing order, and then replace each document identifier with the number subtracted it from its predecessor to form a list of *d*-gaps [2][3]. Although every document identifier is distinct, their *d*-gaps could still form some probability distributions. Some prefix-free coding methods, such as unary code [4], γ code [4], Golomb code [5][6], skew Golomb code [7], and the batched LLRUN code [8], have been proposed for compressing inverted lists by the estimates for these *d*-gaps probability distributions.

The methods for compressing inverted file can yield excellent results if taking the possibility of clustering into account [9]. Although *d*-gap technique is not specially designed for using clustering in compression, many well-known prefix-free coding methods such as skew Golomb code, and the batched LLRUN code can achieve satisfied compression performance via accurate estimates to capture clusters. However, the estimates in these methods are relatively sophisticated, which require more decompression time so that they cannot be applied in real IRSes. Therefore, considering search performance, until now most models such as γ code and Golomb code of *d*-gap technique applied in real IRS are simple and unable to exploit clustering well to achieve good compression [2].

Recently, Moffat and Stuiver have proposed an interpolative coding [9]. Compared with the prefix-free coding methods, the interpolative compression scheme does not require the estimates for the *d*-gaps probability distributions. Based on using clustering with a recursive process of calculating ranges and codes in an interpolative order, superior compression performance yields. However, it is computational expensive due to a stack required in its implementation, which prohibit it from being widely used in the real-world IRSes.

In this paper, we develop a new method based on interpolative compression scheme facilitated by *d*-gap compression scheme is called unique-order interpolative code. It can calculate ranges and codes without using a stack, and hence the decompression time can be greatly reduced. Moreover, it also can exploit clustering well and compress the inverted file efficiently.

This paper is organized as follows. In Section 2, we present the interpolative code that is the most compact method to compress inverted file. In Section 3, we present the unique-order interpolative code. Then we show the quantitative analysis and the simulation results in Section 4 and Section 5. Finally, Section 6 presents our conclusion.

This work was support by National Science Council, ROC: NSC92-2213-E009-065.

2. Interpolative code

Moffat and Stuiver have proposed an elegant compression technique called interpolative code [9]. It can make full use of the clustering in a recursive process of calculating ranges and codes, which demonstrates superior compression performance. In this method, the order as well as lower bound *lo* and upper bound *hi* of every document identifier *x* in an inverted list is calculated and then function Binary_code(*x*, *lo*, *hi*) is called to encode *x*. The detailed algorithm is described in Figure 1. For example, consider the inverted list <7; 1, 2, 5, 6, 8, 10, 13> in a collection of N=20 documents. The full sequence of (*x*, *lo*, *hi*) triples processed by function Binary_code are (6,4,17), (2,2,4), (1,1,1), (5,3,5), (10,8, 19), (8,7,9), and (13,11,20). The simplest encoding mechanism can use binary code to encode *x* and the above triples require 4, 2, 0, 2, 4, 2, and 4 bits, respectively.

The major problem of interpolative coding method is that recursive process is used to calculate the order of every document identifier and its range as well. Although recursive process can be converted to non-recursive one by some well-known techniques [10], the converted codes require a stack to facilitate, which makes the coding and decoding very slow.

However, we observed that calculation of the order and range for every document identifier could be accelerated by using memory to store part of results. Consider a general inverted list IL_t=<f_t; id_1, id_2, …, id_{ft}>, where f_t is the number of documents containing term *t*, id_k<id_{k+1}, and all document identifiers are in the range 1…N. Using the interpolative coding method in Figure 1, for each f_t, we can obtain the full sequence of triples processed for the general list IL_t. Some examples are shown in Table 1. Now, consider a specific inverted list IL=<3; 1, 2, 7> in a collection of N=10 documents, and its triples can be calculated via corresponding triples of f_t = 3 in Table 1. Therefore, the full sequence of triples for IL are (id_2,2,N-1) = (2,2,9), (id_1,1,id_2-1) = (1,1,1), (id_3,id_2+1,N) = (7,3,10). Compared with the method in Figure 1, this one is able to not use a stack to calculate the order and range of each document identifier, which then can save a large amount of time in the calculation.

The corresponding triples of general inverted list IL_t for each f_t can be easily represented as a two-dimensional array I_Triple consisting of f_t rows and 5 columns. An example is clarified in Figure 2. The algorithm in Figure 3 can be used to generate the corresponding I_Triple[f_t][5] for each f_t. If a sub inverted list IL[*index*…*index*+*k*-1] among id_{lo_index}+*lo* and id_{hi_index}+*hi*, Compute_I_Triple (*index*, *k*, *lo_index*, *lo*, *hi_index*, *hi*) can be called to generate the corresponding I_Triple.

Although the procedure Compute_I_Triple in Figure 3 still uses recursive process to generate I_Triple, it can be processed off-line and store corresponding I_Triple of different f_t in memory, which decrease the calculation time of decoding on-line dramatically. After getting corresponding I_Triple in inverted list, we can directly apply binary code in encoding inverted list, which is shown as following:

```
for m:=1 to f_t do
    output bitstring by invoking Binary_code(
            IL[I_Triple[m][1]],
            IL[I_Triple[m][2]]+I_Triple[m][3],
            IL[I_Triple[m][4]]+I_Triple[m][5] );
```

However, this improved method still requires large memory space. This makes it impossible using memory to accelerate coding and decoding of interpolative coding in real IRSes.

```
Algorithm Interpolative_Code(IL, f, lo, hi);
Input: IL (IL[1…f]is a sorted list of f document indetifiers, all
       in the range lo...hi)
Output: bitstring to represent IL[1…f]
begin
  if f = 0 then return;
  if f = 1 then output bitstring by invoking
                    Binary_code(IL[1], lo, hi) and
then return;

  h:=(f div 2)+1;
  f1:=h-1;
  f2:=f-h;
  IL1:=IL[1..(h-1)];
  IL2:=IL[(h+1)…f];
  Output bitstring by invoking Binary_code(IL[h], lo+f1, hi-f2);

  Call interpolative_code(IL1, f1, lo, IL[h]-1);
  Call interpolative_code(IL2, f2, IL[h]+1, hi);
end
```

Figure 1. Interpolative coding.

Table 1. Some examples of the full sequence of triples processed for the general inverted list.

f_t	**The full sequence of triples processed for the general inverted list**
1	(id_1, 1, N)
2	(id_2, 2, N), (id_1, 1, id_2-1)
3	(id_2, 2, N-1), (id_1, 1, id_2-1), (id_3 , id_2+1, N)

The general inverted list as f_t=3:(3;id_1,id_2,id_3), and set id_{f_t+1}= id_4=0 and id_{f_t+2}= id_5=N.
The corresponding triples: (id_2,2,N-1), (id_1,1,id_2-1), (id_3,id_2+1,N).

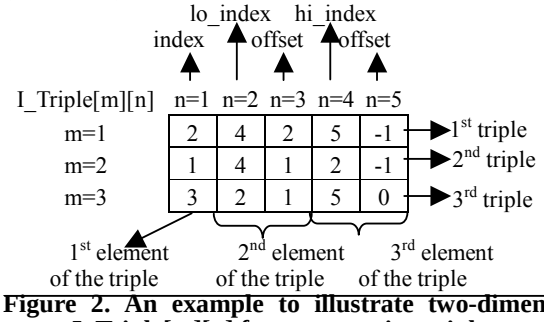

Figure 2. An example to illustrate two-dimensional array I_Triple[m][n] for representing triples.

```
Algorithm Generate_I_Triple(IL, f, N);
Input: IL ( IL[1..f] is a sorted list of f document identifiers, all
       in the range 1...N, and to simplify the algorithm we set
       IL[f+1] to 0, and IL[f+2] to N)
Output: I_Triple[f][5] to represent the triples
begin
   n:=1;   /* n is a global variable*/
   Compute_I_Triple(1, f, f+1, 1, f+2, 0);   /* generate
                                                I_Triple[f][5] */
   return I_Triple;
end

procedure Compute_I_Triple(index, k, lo_index, lo, hi_index, hi)
begin
   if k=0 then return;
   if k=1 then
      I_Triple[n][1]:=index;
      I_Triple[n][2]:=lo_index;
      I_Triple[n][3]:=lo;
      I_Triple[n][4]:=hi_index;
      I_Triple[n][5]:=hi;
      n++;
      return;

   h:=k/2;
   f1:=h;
   f2:=k-h-1;
   I_Triple[n][1]:=h+index;
   I_Triple[n][2]:=lo_index;
   I_Triple[n][3]:=lo+f1;
   I_Triple[n][4]:=hi_index;
   I_Triple[n][5]:=hi-f2;
   n++;

   Compute_I_Triple (index, f1, lo_index, lo, index+h, -1);
   Compute_I_Triple (index+h+1, f2, index+h, 1, hi_index, hi);
end
```

Figure 3. The algorithm for calculating I_Triple.

3. Unique-order interpolative code

We developed a new method called unique-order interpolative code in which only one I_Triple is required for whole coding and decoding process of all inverted lists no matter how many different values of f_t present. Then we introduced loop unrolling to replace I_Triple with constant values. The numbers of memory access for I_Triple therefore can be reduced, which accelerate the whole process.

3.1. Coding method

In an inverted list $IL=<f_t, id_1, id_2, \ldots, id_{ft}>$, f_t is the number of documents containing term t, $id_k<id_{k+1}$ and all document identifiers are in the range $1 \ldots N$. A group size g is first determined. Then according to g, IL is divided into $m=\lceil f/g \rceil$ blocks, each of the first (m-1) blocks has exactly g document identifiers, and the last block has 1... g document identifiers. The first document identifier in each block is defined as boundary pointer, the document identifiers between boundary pointer and boundary pointer as inner pointers, and those in last block except boundary pointer as residual pointers. The boundary pointers and residual pointers can be regarded as an inverted list and the models in d-gap technique with high decoding speed such as γ code and Golomb code to perform compression, while the inner pointers in each block are compressed via interpolative coding method. This new method (Figure 4) contains the constant number of inner pointers g-1, which makes it possible to use only one I_Triple to coding and decoding. Compared with previous interpolative coding method, this new method can let document identifiers store in a fixed order, why we called it unique-order interpolative code. When $f_t \le g$ or m=1 or g=1, we only apply d-gap technique in compression because of no inner pointers present.

Consider a general inverted list $IL_t = <f_t, id_1, id_2, \ldots, id_{ft}>$ encoded by unique-order interpolative code with group size g=4, the IL_t can be represented as $<f_t, id_1, id_5\text{-}id_1\text{-}g\text{-}1, [id_2,id_3,id_4], id_9\text{-}id_5\text{-}g\text{-}1, [id_6,id_7,id_8], id_{13}\text{-}id_9\text{-}g\text{-}1, [id_{10},id_{11},id_{12}], \ldots>$, where id_1, $id_5\text{-}id_1\text{-}g\text{-}1$, $id_9\text{-}id_5\text{-}g\text{-}1$, $id_{13}\text{-}id_9\text{-}g\text{-}1$ are encoded with some prefix-free code and $[id_2,id_3,id_4]$, $[id_6,id_7,id_8]$, $[id_{10},id_{11},id_{12}]$ are encoded with interpolative code. The example list can be further represented (using triple representation in Section 2) as

$<f_t, id_1,$
$id_5\text{-}id_1\text{-}g+1,(id_3,id_1+2,id_5\text{-}2),(id_2,id_1+1,id_3\text{-}1),(id_4,id_3+1,id_5\text{-}1),$
$id_9\text{-}id_5\text{-}g+1,(id_7,id_5+2,id_9\text{-}2),(id_6,id_5+1,id_7\text{-}1),(id_8,id_7+1,id_9\text{-}1),$
$id_{13}\text{-}id_9\text{-}g+1,(id_{11},id_9+2,id_{13}\text{-}2),(id_{10},id_9+1,id_{11}\text{-}1),(id_{12},id_{11}+1,id_{13}\text{-}1)$
,
$\ldots>$.

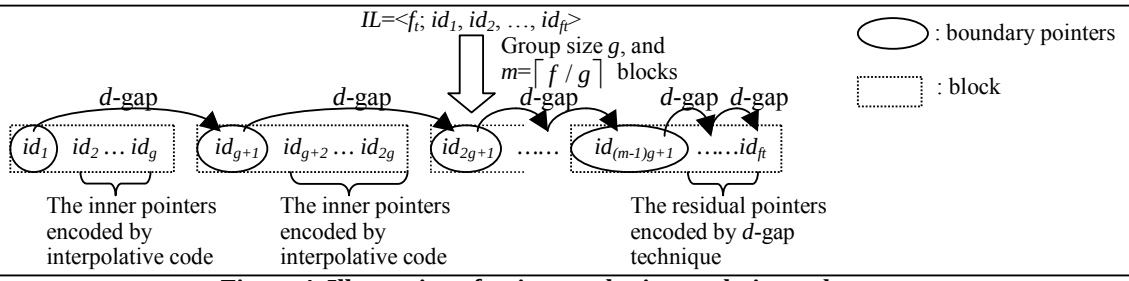

Figure 4. Illustration of unique-order interpolative code

```
Algorithm unique_order_interpolative_code(IL, f, N, g);
Input: IL (IL[1...f] is a sorted list of document numbers,
      all in the range 1...N), and group size g(an integer);
Output: Bitstring (the compressed inverted list IL)
begin
  if  f ≤ g then      // no inner pointers,
                      // encoded by Golomb code
      b := ⌈0.69 × N / f ⌉;
      prev_document_identifier:=0;
      for i:=1 to f
        append Golomb code (
          IL[i]-prev_document_identifier, b) to Bitstring;
        prev_document_identifier:= IL[i];
  else   // encode by unique-order interpolative code
        m = ⌈ f / g ⌉;
        b := ⌈0.69 × N /( f − (m − 1) × (g − 1)) ⌉;

        // encode the first boundary pointer
        append Golomb_code(IL[1], b) to Bitstring;

        // compute I_Triple
        n:=0;   /* n is a global variable */
        I_Triple:=Compute_I_Triple(2, g-1, 1, 1, g+1, -1);

        for i:=0 to (m-2) do
            index:=i×g;

          // encode boundary pointer
          append Golomb_code(
                IL[index+g+1]-IL[index+1]-g+1, b) to Bitstring;

          // encode inner pointers
          for j:=1 to g-1 do
                append Binary_code(
                   IL[index+I_Triple[j][1]],
                   IL[index+I_Triple[j][2]]+I_Triple[j][3],
                   IL[index+I_Triple[j][4]]+I_Triple[j][5]) to
                Bitstring;

        // encode residual pointers
        for i:=(m-1)χg+2 to f
          append Golomb_code(IL[i]-IL[i-1],b) to Bitstring;

  return BitString;
end
```

Figure5. The unique-order interpolative coding method (facilitated by Golomb code)

The indexes in I_Triples for [id_2,id_3,id_4], [id_6,id_7,id_8], [id_{10},id_{11},id_{12}] are differentiated by the value of 4 which is the value of g, while the values of offset are the same. Therefore, only one I-Triple is required in coding and decoding, which accelerate the whole process. If we use Golomb code to encode boundary pointers and residual pointers, this new method can be shown as the above program in Figure 5.

3.2. Discussion

Compared with interpolative code, the new method can avoid the calculations of I-Triples, but a boundary pointer must be set to every certain number of document identifiers, which will enlarge the distance among boundary pointers and reduce compression efficiency. Therefore, a suitable prefix-free code is required to encode boundary pointers and residual pointers to improve compression efficiency.

In this paper, two prefix-free codes are used for boundary pointers and residual pointers, which are Golomb code and γ code. Golomb code is very suitable to encode the *d*-gaps in unique-order interpolative code, since the *d*-gap extracted from every certain amount of document identifiers is roughly the same length. Using γ code is also relatively economical choice when the document identifiers in an inverted list are more concentrated, which can achieve relatively small *d*-gaps.

By the way, once the group size g is determined, the program in Figure 5 can be further accelerated by using loop unrolling technique to replace I_Triple with constant values. For example, group size g=4, the following paragraph program in Figure 5:

```
// encode inner pointers, 8 times memory accesses
// of array IL and I_Triple are required for each
// inner pointer
for j:=1 to g-1 do
    append Binary_code(
        IL[index+I_Triple[j][1]],
        IL[index+I_Triple[j][2]]+I_Triple[j][3],
        IL[index+I_Triple[j][4]]+I_Triple[j][5]) to
        Bitstring;
```

can be converted to

```
// loop unrolling, only 3 times memory accesses of
// array IL are required for each inner pointer
append Binary_code( IL[index+3], IL[index+1]+2,
        IL[index+5]-2) to Bitstring;
append Binary_code( IL[index+2], IL[index+1]+1,
        IL[index+3]-1) to Bitstring;
append Binary_code( IL[index+4], IL[index+3]+1,
        IL[index+5]-1) to Bitstring;
```

In another word, once group size g has been determined, the I_Triple in loop can be replaced with constant values. So the 5(=8-3) times memory accesses for each document identifier can be avoided, which in turn accelerate the whole process.

4. Coding method analysis

To understand the characteristics of unique-order interpolative code, we made two experiments. We used the encoding methods such as Golomb code, skew Golomb code, batched LLRUN code, interpolative code, unique-order interpolative code 1 (group size g=4; boundary pointers and residual pointers by Golomb code), unique-order interpolative code 2 (group size g=4;

boundary pointers and residual pointers by γ code) in compression. In the first experiment (Table 2(a)), f = 1,000,000 gaps were drawn from a geometric distribution and compressed using the six methods. The Golomb code performs the best, since it is a minimum-redundancy code for geometric distribution [6]. Compared with other methods, unique-order interpolative code 1 is not suitable for geometric distribution. But when N/f increases, the performance of unique-order interpolative code 1 becomes better and better. When $N/f \leq 2$, the results of unique-order interpolative code 2 are satisfied.

In the second experiment, for each value of N/f the sequence of f = 1,000,000 geometrically distributed gaps was broken into chunks of 200 contiguous values. The chunks were then placed in groups of five. In the first three chunks of each group, each gap was multiplied by a factor of 0.1; while in the other two chunks each gap was multiplied by a factor of 2.35. This process created artificial cluster of gaps much less than the average, and about 60% of the values are coded into these clusters, but certain overall average gap are retained. This is relatively close to the distribution of real collections. Unique-order interpolative code has good skew geometric distribution (Table 2(b)). Compared with other methods, the compression the compression efficiency of Golomb code is not as good as others which means the compression can be improved with clustering property. When $N/f \leq 32$, we would better use unique-order interpolative code 2 to encode; while $N/f > 32$, we suggest unique-order interpolative code 1. Same with Table 2(a), while the value of N/f becomes larger, the unique-order interpolative code 1 performs better and better. In Table 2(b), the interpolative code can even achieve better compression than self-entropy. This is due to interpolative code does not directly use gap value in encoding, which instead uses minimal binary code to encode after every gap is converted to triples.

5. Experiments

An experimental information retrieval system was implemented to compare various coding methods and techniques. Experiments have been performed on some real-life document collections, and query processing time and the storage requirement for each coding method were measured.

5.1. Document collection and query generation

Three document collections are used in the experiments. The statistics are listed in Table 3. Collection *FBIS* (Foreign Broadcast Information Service) and *LAT* (LA Times), are disk 5 of the TREC-6 collection that is used internationally as a test bed for research in information retrieval techniques [11]. The final collection *TREC* includes the *FBIS* and *LAT* collections.

Since the effective coding methods rely on the clustering, inverted files are developed for each collection with Greedy-NN algorithm [12]. We followed the method [13] to evaluate performance with random queries. For each document collection, 1000 documents are randomly selected to generate a query set. All experiments described in the paper are carried out on Intel P4 2.4GHz machine with 256MB of DDR memory running the Linux

Table 2. Compression results for geometric and skew geometric distributions of f = 1,000,000 gaps: average bits per gap

(a) Geometric distribution

Coding Methods	Average gap (N/f), Geometric Distribution											
	1	2	4	8	16	32	64	128	256	512	1024	2048
Golomb code	1.00	2.33	3.30	4.39	5.43	6.45	7.46	8.47	9.47	10.47	11.47	12.47
Skew Golomb code	1.00	2.53	3.51	4.60	5.64	6.66	7.67	8.68	9.68	10.68	11.68	12.68
Batched LLRUN code	1.00	2.27	3.46	4.50	5.53	6.52	7.52	8.52	9.52	10.52	11.52	12.53
Interpolative code	0.00	2.15	3.45	4.59	5.66	6.69	7.70	8.71	9.71	10.71	11.71	12.72
Unique-order interpolative code 1	3.00	4.19	5.13	5.97	6.76	7.53	8.29	9.06	9.89	10.77	11.68	12.77
Unique-order interpolative code 2	0.25	2.33	3.91	5.31	6.64	7.92	9.19	10.45	11.70	12.96	14.21	15.46
Self-entropy	0.00	2.00	3.24	4.35	5.40	6.42	7.43	8.44	9.44	10.44	11.43	12.43

(b)Skew geometric distribution

Coding Methods	Average gap (N/f), Skewed Distribution											
	1	2	4	8	16	32	64	128	256	512	1024	2048
Golomb	1.40	2.60	3.30	4.29	5.33	6.37	7.39	8.40	9.40	10.40	11.40	12.41
Skew Golomb	1.80	2.31	2.92	3.76	4.80	5.79	6.80	7.82	8.82	9.83	10.83	11.83
Batched LLRUN	1.40	2.31	2.86	3.60	4.61	5.66	6.70	7.71	8.71	9.71	10.70	11.71
Interpolative	0.84	1.53	2.07	2.90	3.97	5.07	6.15	7.19	8.21	9.23	10.23	11.24
Unique-order interpolative code 1	3.60	3.96	4.30	4.80	5.51	6.30	7.11	7.94	8.76	9.60	10.51	11.62
Unique-order interpolative code 2	1.25	1.90	2.47	3.33	4.53	5.88	7.21	8.53	9.81	11.07	12.33	13.60
Self-entropy	0.97	1.77	2.30	3.05	4.06	5.10	6.15	7.18	8.19	9.19	10.19	11.20

Table 3. Statistics of document collections

	Collection		
	FBIS	***LAT***	***TREC***
Documents *N*	130471	131896	262367
Number of terms *F*	72922893	72087460	145010353
Distinct terms *n*	214310	168251	317393
Number of doc. identifiers *f*	28628698	32483656	61112354
Average gap size *N·n/f*	977	683	1363
Total size(MB)	470	475	945

operating system 2.4.12. The size of hard disk is 40GB, and the data transfer rate is about 25MB/sec. Other processes and disk activity were minimized during timing experiments, that is, the machine was under light-load.

5.2. Compression performance of unique-order interpolative code

In this experiment, Golomb code is used to code boundary pointers and residual pointers. As the average gap size in Table 3 is relatively big, Golomb code was used according to section 4.

The result is shown in Table 4, and the metric is the average number of bits per document identifier (BPI), defined as follows:

$$BPI = \frac{\text{The size of the compressed inverted file}}{\text{The total document identfiers } f}.$$

When group size g=4 and g=8, unique-order interpolative code achieves good compression. Considering about the convenience of application, we suggest to select group size g=4. In the following experiments, without specified indication, encoding of unique-order interpolative code is facilitated by Golomb code with group size g=4.

Table 4. Compression performance (BPI) of unique-order interpolative code with different group size *g*

Group Size *g*	Collection		
	FBIS	*LAT*	*TREC*
g=1	5.27	5.31	5.49
g=2	4.84	4.91	4.99
g=3	4.80	4.89	4.94
g=4	4.66	4.74	4.78
g=5	4.71	4.80	4.82
g=6	4.71	4.79	4.81
g=7	4.65	4.74	4.75
g=8	4.59	4.68	4.69
g=9	4.64	4.72	4.73
g=10	4.67	4.75	4.76

5.3. Compression performance of different coding methods

We now show the effectiveness of the different coding methods. The metric is BPI defined in Section 5.2. The results (Table 5) showed that: 1.Ihe compression efficiency of γ code and Golomb code is relatively low because of the simple models they use; 2.The compression efficiency of batched LLRUN code, skew Golomb code, interpolative code, unique order interpolative code is relatively good because they use clustering to compress; 3.It is confirmed that unique-order interpolative code can take advantage of clustering property. Using clustering, the compression efficiency of unique-order interpolative code is only lower than interpolative code.

Table 5. Compression performance of different coding methods

Coding Methods	Collection		
	FBIS	*LAT*	*TREC*
γ code	5.38	5.63	5.63
Golomb code	5.27	5.31	5.49
Batched LLRUN code	4.63	4.78	4.84
Skew Golomb code	5.04	5.07	5.10
Interpolative code	4.58	4.65	4.62
Unique-order interpolative code	4.66	4.74	4.78

5.4. Search performance of different coding methods

The query processing time includes (1) disk access time, (2) decompression time, and (3) document identifiers comparison time. According to the experimental results, disk access time and decompression time occupy more than 90% of query processing time. And document identifier comparison time is not affected by coding method. Therefore performance metric is defined as

Processing Time (**PT**) = *Disk Access Time* (**AT**) + *Decompression Time* (**DT**).

Based on Execution Time of Golomb code, speed-up (**SP**) of each test collection in various coding methods was calculated.

In this experiment, all decoding mechanisms has undergone the best process as following: 1.Remove nested function calls; 2.Replace subroutines with macros; 3.Replace log function with fast bit-shift; 4.Set compiler optimization flag, such as –O, -O1, -O2; 5.Take advantage of 32-bit CPU characteristics: use 32-bit long register to hold and process compressed data.

Except of the above process, the Huffman code of batched LLRUN code uses canonical prefix codes to accelerate the process [14]. While interpolative coding method uses certain well-known technique to convert recursive process to non-recursive process, but it still requires an explicit stack in practice [10].

The results are shown in Table 6. Compared with other traditional encoding methods such as batched LLRUN code, skew Golomb code, interpolative code, the processing time of γ code, Golomb code is shorter. Although the processing time of γ code is shorter than Golomb code, its compression efficiency in large-scale

databases such as *FBIS, LAT, TREC* is worse than Golomb code. Considering about search performance and compression ratio in IRS, Golomb code is used to coding and compression of inverted files. And the query processing time of unique-order interpolative code is shortest.

Table 6. Average query processing time of different coding methods

Coding Methods		Collection		
		FBIS	*LAT*	*TREC*
γ code	**PT**(us)	2596	2662	4800
	SP	1.07	1.07	1.08
Golomb code	**PT**(us)	2789	2851	5174
	SP	1.00	1.00	1.00
Batched LLRUN code	**PT**(us)	3137	3212	5700
	SP	0.89	0.89	0.91
Skew Golomb code	**PT**(us)	2841	2960	5192
	SP	0.98	0.96	1.00
Interpolative code	**PT**(us)	4214	4362	7687
	SP	0.66	0.65	0.67
Unique-order interpolative code	**PT**(us)	2493	2494	4603
	SP	1.12	1.14	1.12

The results in Table 7 show that the ratio of disk access time to decompression time occupied is relatively equal for Golomb code and γ code. Therefore, a good encoding method must be characterized by high compression efficiency as well as high decoding velocity. The unique interpolative code is such kind of good method. So it can take place of Golomb code in IRS.

Table 7. The ratio of disk access time and decompression time of different coding methods

Coding Methods		Collection		
		FBIS	*LAT*	*TREC*
γ code	**AT**(%)	43	44	45
	DT(%)	57	56	55
Golomb code	**AT**(%)	46	46	47
	DT(%)	54	54	53
Batched LLRUN code	**AT**(%)	36	36	37
	DT(%)	64	64	63
Skew Golomb code	**AT**(%)	39	38	40
	DT(%)	61	62	60
Interpolative code	**AT**(%)	24	23	25
	DT(%)	76	77	75
Unique-order interpolative code	**AT**(%)	43	43	44
	DT(%)	57	57	56

6. Conclusion

This paper proposes a novel coding method, unique-order interpolative code, for compressing inverted files in IRSes. The new method is much easier to implement than the interpolative code, however, it is also suited to situations in which clustering is anticipated, and experiments with the inverted files of three test databases show the method to yield superior performance for both fast querying and space-efficient indexing. This work shows a feasible way to build a fast and space-economical IRS.

References

[1] W. B. Frankes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[2] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing on Documents and Images*. Morgan Kaufmann, San Francisco, California, second edition, 1999.

[3] A. Moffat and J. Zobel, "Parameterised compression for sparse bitmaps," *Proceedings of 15th annual international ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp. 274-285, Copenhagen, June 1992. ACM Press, New York.

[4] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, *Vol*. IT-21, *No*. 2, pp. 194-203, 1975.

[5] S. W. Golomb, "Run Length Encoding," *IEEE Transactions on Information Theory*, *Vol.* IT-12, *No.* 3, pp. 399-401, 1966.

[6] R. G. Gallager and D. C. Van Voorhis, "Optimal source codes for geometrically distributed alphabets," *IEEE Transactions on Information Theory*, *Vol.* IT-21, *No.* 2, pp. 228-230, Mar. 1975.

[7] J. Teuhola, "A Compression method for clustered bit-vectors," *Information Processing Letters*, *Vol.* 7, *No.* 6, pp.308-311, Oct. 1978.

[8] A. S. Fraenkel and S. T. Klein, "Novel Compression of sparse bit-string—Preliminary report," in: A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, *Vol.* 12, NATO ASI Serials F, pp. 169-183, Berlin, 1985. Springer-Verlag.

[9] A. Moffat, and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, *Vol.* 3, *No*. 1, pp. 25-47, July 2000.

[10] A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein, *Data structures using C*. Englewood CLiffs, N.J. 07632, Prentice-Hall, 1990.

[11] E. Voorhees, and D. Harman, "Overview of the sixth text retrieval conference (TREC-6)," *Proceedings of the 6th Text Retrieval Conference*, pp. 1-24, 1998. NIST Special Publication 500-240.

[12] Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shaun and Chung-Ping Chung, "Inverted file compression through document identifier reassignment," *Information Processing and Management*, *Vol*. 39, *No*. 1, pp. 117-131, 2003.

[13] A. Moffat and J. Zobel, "self-indexing inverted files for fast text retrieval," *ACM Transactions on Information Systems*, *Vol*. 14, *No*. 4, pp. 349-379, 1996.

[14] A. Turpin, "Efficient prefix coding," PhD thesis, University of Melbourne, 1998.

計畫成果自評

本計畫規劃了一系列的研究，探討如何以最小的資源成本建置符合需求的叢集式資訊檢索系統，並透過動態的資源管理機制，讓系統在面對各種不同的外界環境時，能夠動態地調整系統的資源配置，將系統的資源發揮最大的效能，期以最小的資源成本提供使用者一個高效能和高服務品質的資訊檢索環境。在本年度的研究中，為了能夠有效增進系統效能與儲存空間利用率，我們發展了一個新的轉置檔案壓縮機制。實驗證明此一壓縮機制的壓縮率好，解碼速度快，並且可以支援 self-indexing strategy，使得資料查詢的處理速度大幅的提升。此一研究成果已經在 ITCC2004 國際會議上發表，並投稿到 Information processing and management 國際期刊等待審查。未來在此研究基礎上，本計畫將持續探討叢集式資訊檢索系統的負載、快取與資料管理方法，期能以最小的成本滿足給定的執行效能需求。並發展一資訊檢索雛型系統，實作並驗證所提各項技術之可行性。