Fast String Sorting using Order Preserving Compression

ALEJANDRO LÓPEZ-ORTIZ
University of Waterloo
MEHDI MIRZAZADEH
University of Waterloo
MOHAMMAD ALI SAFARI
Universiy of British Columbia
and
HOSSEIN SHEIKHATTAR
University of Waterloo

We give experimental evidence for the benefits of order preserving compression in sorting algorithms. While in general any algorithm might benefit from compressed data due to reduced paging requirements, we identified two natural candidates that would further benefit from order preserving compression, namely string-oriented sorting algorithms and word-RAM algorithms for keys of bounded length. The word-RAM model has some of the fastest known sorting algorithms in practice. These algorithms are designed for keys of bounded length, usually 32 or 64 bits, which limits their direct applicability for strings. One possibility is to use an order preserving compression scheme, so that a bounded-key-length algorithm can be applied. For the case of standard algorithms we took what is considered to be the among the fastest non-word RAM string sorting algorithms, Fast MKQSort, and measured its performance on compressed data. The Fast MKQSort algorithm of Bentley and Sedgewick is optimized to handle text strings. Our experiments show that order compression techniques results in savings of approx. 15% over the same algorithm on non-compressed data. For the word-RAM we modified Andersson's sorting algorithm to handle variable length keys. The resulting algorithm is faster than the standard Unix sort by a factor of 1.5x. Lastly we used an order preserving scheme that is within a constant additive term of the optimal Hu-Tucker but requires linear time rather than $O(m \log m)$ where $m = |\Sigma|$ is the size of the alphabet.

Categories and Subject Descriptors: E.2 [**Data Storage**]: Contiguous Representations,Object Representations; E.4 [**Coding and Information Theory**]: Data Compaction and Compression; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; H.3.2 [**Information Storage**]: File Organization; H.3.2 [**Information Search and Retrieval**]: Search process

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Order preserving compression, sorting, word-RAM, unit-cost RAM

Alejandro López-Ortiz, Mehdi Mirzazadeh and Hossein SheikhAttar are at the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ont., Canada, N2L 3G1. Email: {alopez-o, mmirzazadeh,mhsheikhattar}@uwaterloo.ca.
Mohammad Ali Safari is at the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4. Email: safari@cs.ubc.ca.

1. INTRODUCTION

In recent years, the size of corporate data collections has grown rapidly. For example, in the mid-1980s, a large text collection was in the order of 500 MBytes. Today, large text collections are over a thousand times bigger. At the same time, archival legacy data that used to sit in tape vaults is now held on-line in large data-warehouses and accessed regularly. Data storage companies such as EMC have emerged to serve this need for data storage, with market capitalizations that presently rival that of all but the largest PC manufacturers.

Devising algorithms for these massive data collections requires novel techniques. Because of this, over the last ten years there has been renewed interest in research on indexing techniques, string matching algorithms and very large database management systems among others.

Consider, for example, a corporate setting, such as a bank, with a large collection of archival data, say, a copy of every bank transaction ever made. Data is stored in a data warehouse facility and accessed periodically, albeit perhaps somewhat unfrequently. Storing the data requires a representation that is succinct, amenable to arbitrary searches and supports efficient *random* access.

Aside from savings in storage, a no less important advantage of a succinct representation of archival data is a resulting improvement in performance of sorting and searching operations. This improvement is twofold: First, in general almost any sorting and searching algorithm benefits from operating on smaller keys, as this leads to a reduced number of page faults as observed by Moura et al. [Moura et al. 1997]. Second, benefits can be derived from using a word-RAM (unit-cost RAM) sorting algorithm, such as that of Andersson [Andersson 1994; Andersson and Nilsson 1998]. This algorithm sorts n w-bits keys on a unit-cost RAM with word size w in time $O(n\sqrt{\log n})$. As one can expect, in general this algorithm cannot be applied to strings as the key length is substantially larger than the word size. However if all or most of the keys can be compressed to below the word size then this algorithm can be applied— with ensuing gains in performance.

There are many well known techniques for compressing data, however most of them are not order preserving and do not support random access to the data as the decoding process is inherently sequential [Bell et al. 1990; Knuth 1997]. Hence, it is important that the compression technique be static as well as order preserving. This rules out many of the most powerful compression techniques, such as those based on the Ziv-Lempel method [Ziv and Lempel 1978], which are more attuned to compression of long text passages in any event (we note, however, that it is possible to implement certain search operations on Lempel-Ziv encoded text as shown by Farach et al. and Kärkkäinen et al. [Farach and Thorup 1998; Kärkkäinen and Ukknonen 1996]). Also, the need to preserve order eliminates many dictionary techniques such as Huffman codes [Knuth 1997; Antoshenkov 1997].

Hence we consider a special type of static code, namely, *order-preserving compression schemes*, which are similar to Huffman codes. More precisely, we are given an alphabet Σ with an associated frequency p_i for each symbol $s_i \in \Sigma$. The compression scheme $E : \Sigma \rightarrow \{0, 1\}^*$ maps each symbol into an element of a set of prefix free strings over $\{0, 1\}^*$. The goal is to minimize the entropy $\sum_{s_i \in \Sigma} p_i |E(s_i)|$ with the added condition that if $s_i < s_j$ then $E(s_i) < E(s_j)$ in the lexicographic ordering of $\{0, 1\}^*$. This is in contrast to Huffman codes which in general do not preserve the order of the initial alphabet. Such a scheme is known as *order preserving*.

Optimal order-preserving compression schemes were introduced by Gilbert and Moore [Gilbert and Moore 1959] who gave an $O(n^3)$ algorithm for computing the optimal code. This was later improved by Hu and Tucker, who gave a $\Theta(n \log n)$ algorithm, which is optimal [Hu 1973]. In this paper we use a linear time encoding algorithm that approximates the optimal order preserving compression scheme to within a constant additive term to produce a compressed form of the data. The savings are of significance when dealing with large alphabets which arise in applications such as DNA databases and compression on words. We test the actual quality of the compression scheme on real string data and obtain that the compressed image produced by the linear algorithm is within 0.4% and 5.2% of the optimal, in the worst case. The experiments suggest that the compression ratio is, in practice, much better than what is predicted by theory.

Then, using the compressed form we test the performance of the sorting algorithm against the standard Unix sort in the Sun Solaris OS. Using data from a 1GB world wide web crawl, we study first the feasability of compressing the keys to obtain 64 bit word length keys. In this case we obtain that only 2% of the keys cannot be resolved within the first 64 significant bits. This small number of keys are flagged and resolved in a secondary stage. For this modified version of Andersson's we report a factor of 1.5x improvement over the timings reported by Unix sort.

As noted above, an important advantage of order preserving compression schemes is that they support random access. As such we consider as a likely application scenario that the data is stored in compressed format. As an example, consider a database management system (DBMS). Such systems often use sorting as an intermediate step in the process of computing a `join` statement. The DBMS would benefit from an order preserving compression scheme first by allowing faster initial loading (copying) of the data into memory and second by executing a sorting algorithm tuned for compressed data which reduces both processing time and the amount of paging required by the algorithm itself if the data does not fit in main memory. The contribution of each of these aspects is highlighted in Tables V and VI.

The paper is laid out as follows. In Section 2 we introduce the linear time algorithm for encoding and observe that its approximation term follows from a theorem of Bayer [Bayer 1975]. In Section 3 we compare the compression ratio empirically for string data using the Calgary corpus. In Section 4 we compare the performance of sorting algorithms aided by order preserving data compression.

2. ORDER PRESERVING COMPRESSION

We consider the problem of determining code-words (encoded forms) such that the compression ratio is as high as possible. Code-words may or may not have the prefix property. In the prefix property case, the problem reduces to finding an optimum alphabetic binary tree.

Problem Definition Formally, the problem of finding optimal alphabetic binary trees can be stated as follows: Given a sequence of n positive weights $w_1, w_2, \cdots, w_n$, find a binary tree in which all weights appear in the leaves such that

—The weights on the leaves occur in order when traversing the tree from left to right. Such a tree is called an *alphabetic tree.*

—The sum $\sum_{1 \leq i \leq n} w_i l_i$ is minimized, where l_i is the depth (distance from root) of the ith leave from left. If so, this is an *optimal alphabetic tree*

If we drop the first condition, the problem becomes the well-known problem of building Huffman trees, which is known to have the same complexity as sorting.

Previous Work Mumey introduced the idea of finding optimal alphabetic binary tree in linear time for some special classes of inputs in [Mumey 1992]. One example is a simple case solvable in $O(n)$ time when the values of the weights in the initial sequence are all within a term of two. Mumey showed that the region-based method, described in [Mumey 1992], exhibits linear time performance for a significant variety of inputs. Linear time solutions were discovered for the following special cases: when the input sequence of nodes is sorted sequence, bitonic sequence, weights exponentially separated, and weights within a constant factor, (see [Larmore and Przytycka 1998]).

Moura et al. considered the benefits of constructing a suffix tree over compressed text using an order preserving code [Moura et al. 1997]. In their paper they observe dramatic savings in the construction of a suffix tree over the compressed text.

2.1 A Simple Linear Approximation Algorithm

Here, we present an algorithm which creates a compression dictionary in linear time on the size of the alphabet and whose compression ratio compares very favourably to that of optimal algorithms which have $\Omega(n \log n)$ running time, where n is the number of symbols or tokens for the compression scheme. For the purposes of presentation we refer to the symbols or tokens as *characters* in an *alphabet* Σ. In practice these "characters" might well correspond to, for example, entire English words or commonly occurring three or four letter combinations. In this case the "alphabet" can have tens of thousands of tokens, and hence the importance of linear time algorithms for creating the compression scheme.

The idea of the proposed algorithm is to divide the set of weights into two almost equal size subsets and solve the problem recursively for them. As we show in section 2.4, this algorithm finds a compression scheme within an additive term of 2 bits of the average code length found by Huffman or Hu-Tucker algorithms.

2.2 Algorithm

Let $w_1, w_2, \cdots, w_n$ be the weights of the alphabet characters or word codes to be compressed in alphabetical order. The procedure *Make(i, j)* described below, finds a tree in which tokens with weights $w_i, w_{i+1}, \cdots, w_j$ are in the leaves:

procedure *Make*$(i,\ j)$

(1) if $(i == j)$ return a tree with one node containing w_i.
(2) Find k such that $|(w_i + w_{i+1} + ... + w_k) - (w_{k+1} + ... + w_j)|$ is minimum.
(3) Let T_1 = Make$(i,\ k)$ and T_2 = Make$(k+1,\ j)$
(4) Return tree T with left subtree T_1 and right subtree T_2.

In the next two subsections we study (a) the time complexity of the proposed algorithm and (b) bounds on the quality of the approximation obtained.

2.3 Time Complexity

First observe that, aside from line 2, all other operations take constant time. Hence so long as line 2 of the algorithm can be performed in logarithmic time, then the running time $T(n)$ of the algorithm would be given by the recursion $T(n) = T(k) + T(n-k) + O(\log k)$, and hence, $T(n) = O(n)$. Therefore, the critical part of the algorithm is line 2: how to divide the set of weights into two subsets of almost the same size in logarithmic time.

Suppose that for every k, the value of $a_k = b_i + w_i + w_{i+1} + \cdots + w_k$ is given for all $k \geq i$ where b_i is a given integer to be specified later. Notice that the a_j's form an increasing sequence as $a_i < a_{i+1} < \cdots < a_j$. Now the expression in line 2 of the algorithm can be rewritten as follows:

$$|(w_i + w_{i+1} + ... + w_k) - (w_{k+1} + ... + w_j)| = |a_j - 2a_k + b_i|.$$

So, given the value of a_k, for all k, one can easily find the index u for which $|a_j - 2a_u + b_i|$ is minimum using a variation of a one sided binary search, known as galloping. Define $a_k := \sum_{i=1}^{k} w_k$ and hence $b_i = a_{i-1} = \sum_{\ell=1}^{i-1} w_\ell$ and modify the algorithm as follows:

procedure *Make*$(i,\ j,\ b)$

(1) if $(i == j)$ return a tree with one node containing w_i.
(2) let = Minimize$(i,\ j,\ b,\ 1)$.
(3) Let T_1 = Make$(i,\ k,\ b)$ and T_2 = Make$(k+1,\ j,\ a_k)$
(4) Return tree T with left subtree T_1 and right subtree T_2.

where the minimize procedure is a one sided binary search for the element closest to zero in the sequence $\{a_j - 2a_k - 2b\}_k$. More precisely:

procedure *Minimize*$(i,\ j,\ b,\ g)$

(1) if $(j - i \leq 1)$ return $\min\{|a_j - 2a_i - 2p|, |a_j + 2b|\}$.
(2) let $\ell = i + g$, $u = j - g$.
(3) if $(a_j - 2a_\ell - 2b) > 0$ and $(a_j - 2a_u - 2b) < 0$ then return Minimize$(i,j,b,2g)$.
(4) if $(a_j - 2a_\ell - 2b) < 0$ then return Minimize$(\ell - g/2,\ell,b,1)$.
(5) if $(a_j - 2a_u - 2b) > 0$ then return Minimize$(u,u + g/2,b,1)$.

The total time taken by all calls to *Minimize* is given by the recursion $T(n) = T(k) + T(n-k) + \log k$, if $k \leq n/2$ and $T(n) = T(k) + T(n-k) + \log(n-k)$ otherwise, where n is the number of elements in the entire range and k is the position of the element found in the first call to *Make*. This recursion has solution $T(n) \le 2n - \log n - 1$ as can easily be verified:

$$T(n) \;=\; T(k) + T(n-k) + \log k \;\le\; 2n - 1 - \log(n-k) - 1 \le 2n - 1 - \log n$$

when $k \le n/2$. The case $k > n/2$ is analogous.

To compute total time, we have that, at initialization time, the algorithm calculates a_k for all k and then makes a call to Make(1, n, 0). The total cost of line 2 is linear and hence the entire algorithm takes time $O(n)$.

2.4 Approximation bounds

Recall that a binary tree T can be interpreted as representing a coding for symbols corresponding to its leaves by assigning 0/1 labels to left/right branches, respectively. Given a tree T and a set of weights associated to its leaves, we denote as $E(T)$, the expected number of bits needed to represent each of these symbols using codes represented by the tree. More precisely, if T has n leaves with weights $w_1, \ldots, w_n$ and depths $l_1, \ldots, l_n$, then

$$E(T) = \sum_{i=1}^{n} \frac{w_i\, l_i}{W(T)},$$

where $W(T)$ is defined as the total sum of the weights $\sum_{i=1}^{n} w_i$. Note that $W(T) = 1$ for the case of probability frequency distributions.

THEOREM 2.1. *Let T be the tree generated by our algorithm, and let T_{OPT} be the optimal static binary order preserving code. Then*

$$E(T) \le E(T_{OPT}) + 2.$$

This fact can be proven directly by careful study of the partition mechanism depending on how large the central weight w_k is. However we observe that a much more elegant proof can be derived from a rarely cited work by Paul Bayer. Consider a set of keys $k_1, \ldots, k_n$, with probabilities $p_1, \ldots, p_n$ for successful searches and $q_0, q_1, \ldots, q_n$ for unsuccessful searches. Let $H = \sum_{i=1}^{n} -p_i \lg p_i + \sum_{i=0}^{n} -q_i \lg q_i$ denote the entropy of the associated probability distribution. Observe that from Shannon's source coding theorem [Shannon 1948], we know that $H \le E(T)$ for any tree T.

Definition 2.2. A *weight balanced* tree is an alphabetic binary search tree constructed recursively from the root by minimizing the difference between the weights of the left and right subtrees.

That is, a weight balanced tree minimizes $|W(L) - W(R)|$ in a similar fashion to procedure Make above.

THEOREM 2.3 [BAYER 1975]. *Let S_{OPT} denote the optimal alphabetic binary search tree, with keys in internal nodes and unsuccessful searches in external nodes. Let S denote a weight balanced tree on the same keys, then*

$$E(S) \le H + 2 \le E(S_{OPT}) + 2$$

With this theorem at hand we can now proceed with the original proof of Theorem 2.1.

Table I. Comparison of the three algorithms

Alphabet size n	Linear / Huffman	Hu-Tucker / Huffman	Linear / Hu-Tucker
$n = 26$ (10000 tests)	1.1028	1.0857	1.0519
$n = 256$ (10000 tests)	1.0277	1.0198	1.0117
$n = 1000$ (3100 tests)	1.0171	1.0117	1.0065
$n = 2000$ (1600 tests)	1.0147	1.0100	1.0053
$n = 3000$ (608 tests)	1.0120	1.0089	1.0038

Proof of Theorem 2.1. We are given a set of symbols s_i and weights w_i with $1 \leq i \leq n$. Consider the weight balanced alphabetical binary search tree on $n-1$ keys with successful search probabilities $p_i = 0$ and unsuccessful search probabilities $q_i = w_{i-1}$. Observe that there is a one-to-one mapping between alphabetic search trees for this problem and order preserving codes. Moreover the cost of the corresponding trees coincide. It is not hard to see that the tree constructed by Make corresponds to the weight balanced tree, and that the optimal alphabetical binary search tree S_{OPT} and the optimum Hu-Tucker code tree T_{OPT} also correspond to each other. Hence from Bayer's theorem we have

$$E(T) \leq H + 2 \leq E(S_{OPT}) + 2 = E(T_{OPT}) + 2$$

as required. □

This shows that in theory the algorithm proposed is fast and has only a small performance penalty in terms of compression over both the optimal encoding method and the information theoretical lower bound given by the entropy.

3. EXPERIMENTS ON COMPRESSION RATIO

In this section we compare experimentally the performance of the algorithm in terms of compression against other static compression codes. We compare three algorithms: *Huffman*, *Hu-Tucker* and our algorithm on a number of random frequency distributions. We compared alphabets of size n, for variable n. In the case of English this corresponds to compression on words, rather than on single characters. Each character was given a random weight between 0 and $100,000$, which is later normalized. The worst case behavior of our algorithm in comparison with *Hu-Tucker* and *Huffman* algorithms is shown in Table I. For each sample we calculated the expected number of bits required by each algorithm on that frequency distribution and reported the ratio least favourable among those reported.

We also compared the performance of the proposed linear time algorithm with Huffman and Hu-Tucker compression using the Calgary corpus, a common benchmark in the field of data compression. This is shown in Table II. We report both the compression ratio of each of the solutions as well as the comparative performance of the linear time solution with the other two well known static methods. As we can see, the penalty on the compression factor of the linear time algorithm over Huffman, which is not order preserving, or Hu-Tucker, which takes time $O(n \log n)$, is minimal.

It is important to observe that for the data set tested the difference between the optimal Hu-Tucker and the linear compression code was in all cases below 0.2 bits, which is much less than the worst case additive term of 2 predicted by Bayer's theorem.

Table II. Comparison using the Calgary Corpus

file	size (in bits)	Huff.	Lin.	H-T	Lin./Huff.	Lin./H-T	H-T/Huff.
bib.txt	890088	65%	68%	67%	1.0487	1.0140	1.0342
book1.txt	6150168	57%	61%	59%	1.0727	1.0199	1.0518
book2.txt	4886848	60%	63%	62%	1.0475	1.0159	1.0310
paper1.txt	425288	62%	65%	64%	1.0378	1.0075	1.0301
paper2.txt	657592	57%	60%	60%	1.0520	1.0098	1.0418
paper3.txt	372208	58%	61%	60%	1.0421	1.0099	1.0318
paper4.txt	106288	59%	63%	61%	1.0656	1.0321	1.0324
paper5.txt	95632	62%	65%	64%	1.0518	1.0151	1.0361
paper6.txt	304840	63%	66%	64%	1.0495	1.0198	1.0290
progc.txt	316888	65%	68%	66%	1.0463	1.0315	1.0143
progl.txt	573168	59%	63%	61%	1.0637	1.0324	1.0302
progp.txt	395032	61%	64%	63%	1.0583	1.0133	1.0443
trans.txt	749560	69%	72%	70%	1.0436	1.0243	1.0187
news.txt	3016872	65%	67%	67%	1.0403	1.0103	1.0296
geo	819200	70%	72%	71%	1.0173	1.0098	1.0074
obj1	172032	74%	76%	75%	1.0220	1.0149	1.0070
obj2	1974512	78%	80%	80%	1.0280	1.0103	1.0175
pic	4105728	20%	21%	21%	1.0362	1.0116	1.0242

4. STRING SORTING USING A WORD RAM ALGORITHM

In this section we compare the performance of sorting on the compressed text against the uncompressed form (as in [Moura et al. 1997]) including Bentley and Sedgewicks's FastSort [Bentley and Sedgewick 1997] as well as Andersson's word-RAM sort [Andersson 1994]. Traditionally word RAM algorithms operate on unbounded length keys, such as strings, by using radix/bucket sort variants which iteratively examine the keys [Andersson and Nilsson 1994]. In contrast our method uses order preserving compression to first reduce the size of the keys, then sort using fixed key size word RAM algorithms [Andersson et al. 1995], sorting keys into buckets. We observed experimentally that this suffices to sort the vast majority of the strings when sorting 100MB files of web crawled text data. In this case each of the buckets contained very few elements in practice. We also tested the algorithms on the Calgary corpus, which is a standard benchmark in the field of text compression.

We consider the use of a word RAM sorting algorithm to create a dictionary of the words appearing in a given text. The standard word RAM algorithms have as a requirement that the keys fit within the word size of the RAM machine being used. Modern computers have word sizes of 32 or 64 bits. In this particular example we tested Andersson's 32 bit implementation [Andersson and Nilsson 1998] of the $O(n \log\log n)$ algorithm by Andersson et al. [Andersson et al. 1995]. We also tested the performance of Bentley and Sedgewick's MKQSort [Bentley and Sedgewick 1997] running on compressed data using order preserving compression. The algorithm is a straightforward implementation of the code in [Bentley and Sedgewick 1997].

Observe that one can use a word RAM algorithm on keys longer than w bits by initially sorting on the first w bits of the key and then identifying "buckets" where two or more strings are "tied", i.e. share the first w bits. The algorithm proceeds

Table III. Percentage of buckets multiply occupied (Web crawl).

File	Size in tokens	% words sharing a bucket		
		Uncompressed	Hu-Tucker	Linear
alphanumeric	515277	16%	2%	2%
alpha only	373977	21%	3%	3%

recursively on each of these buckets until there are no further ties. This method is particularly effective if the number of ties is not too large.

To study this effect, we consider two word dictionaries and a text source. One is collected from a 1 GB crawl of the World Wide Web, the second from all the unique words appearing in the Calgary corpus and the last one is a more recent 3.3 GB crawl of the World Wide Web. This is motivated by an indexing application for which sorting a large number of words was required. In principle, the result is equally applicable to other settings such as sorting of alphanumeric fields in a database.

In the case of the web crawl we considered two alternative tokenization schemes. The first one tokenizes on alphanumeric characters while the second ignores numbers in favour of words only. Table III shows the number of buckets that have more than one element after the first pass in the uncompressed and the compressed form of the text. We report both the figure for the proposed linear algorithm and for the optimal Hu-Tucker scheme. Observe the dramatic reduction in the number of buckets that require further processing in the compressed data.

In fact the numbers of ties in the compressed case is sufficiently small that aborting the recursion after the first pass and using a simpler sorting algorithm on the buckets is a realistic alternative. In comparison, in the uncompressed case the recursion reaches depth three in the worst case before other types of sorting become a realistic possibility.

In the case of the Calgary corpus, the number of buckets with ties in the uncompressed form of the text ranged from 3% to 8%. After compressing the text the number of ties, in all cases, rounded to 0.0%. The specific figures for a subset of the Calgary corpus are shown in Table 4.

Notice that in all cases the performance of the optimal Hu-Tucker algorithm and the linear algorithm is comparable. We should also emphasize that while the tests in this paper used compression on alphanumeric characters only, the compression scheme can be applied to entire words (see for example [Mumey 1992]). In this case, the size of the code dictionary can range in the thousands of symbols which makes the savings of a linear time algorithm particularly relevant.

Lastly we considered a series of ten web crawls from Google, each of approximately 100MB in size (3.3GB in total). In this case we operate under the assumption that the data is stored in the suitable format to the corresponding algorithm. We posit that it is desirable to store data in compressed format, as this results also in storage savings while not sacrificing searchability due to the order preserving nature of the compression. We tokenized and sorted each of these files to create a dictionary, a common preprocessing step for some indexing algorithms. The tokenization was performed on non-alphanumeric characters. For this test we removed tokens larger than 32 bits from the tokenized file. In practice, these tokens would be sorted using a second pass, as explained before. We first studied the benefits of

Table IV. Percentage of buckets multiply occupied on the text subset of the Calgary Corpus.

File	% Percentage words tied		
	Uncompressed	Hu-Tucker	Linear
paper1	5%	0%	0%
paper2	4%	0%	0%
paper3	6%	0%	0%
paper4	4%	0%	0%
paper5	3%	0%	0%
paper6	4%	0%	0%
book1	3%	0%	0%
book2	8%	0%	0%
bib	7%	0%	0%

Table V. CPU time (in seconds) as reported by the Unix `time` utility.

Algorithm	Data 1	Data 2	Data 3	Average	Variance
QSort	2.41	2.29	2.33	2.33	0.04
QSort on Compressed	2.17	2.18	2.20	2.20	0.05
Andersson	0.80	0.82	0.81	0.81	0.01
Fast Sort	0.88	0.89	0.86	0.88	0.02
Fast Sort on Compressed	0.73	0.75	0.75	0.74	0.02
Binary Search	1.27	1.29	1.26	1.28	0.02
Binary Search on Compressed	1.08	1.09	1.09	1.09	0.02

order preserving compression alone by comparing the time taken to sort the uncompressed and compressed forms of the text. The tokens were sorted using the Unix `sort` routine, Unix `qsort`, Fast MKQSort and Andersson's sort algorithm [Andersson and Nilsson 1998]. Table V shows a comparison of CPU times among the different sorting algorithms, while table VI shows the comparison in performance including I/O time for copying the data into memory. Note that there are observed gains both in CPU time alone and in CPU plus I/O timings as a result of using the data compressed form.

We report timings in individual form for the first three crawls as well as the average sorting time across all 10 files together with the variance. On the data provided Fast MKQSort is the best possible choice with the compressed variant being 20% faster than the uncompressed form. These are substantial savings for such a highly optimized algorithm.

While in this case we focused on key lengths below w bits, the savings from compression can be realized by most other sorting, searching or indexing mechanisms, both by the reduction of the key length field and by the reduced demands in terms of space. To emphasize, there are two aspects of order preserving compression which have a positive impact on performance. The first is that when comparing two keys byte-per-byte, we are now in fact comparing more than one key at once, since compressed characters fit at a rate of more than one per byte. Secondly the orginal data size is reduced. This leads to a decrease in the amount of paging to external memory, which is often the principal bottleneck for algorithms on large data collections.

Table VI. Total system time (in seconds) as reported by the Unix `time` utility.

Algorithm	Data 1	Data 2	Data 3	Average	Variance
Unix Sort	5.53	5.40	5.30	5.41	0.07
Unix Sort Compressed	5.43	5.43	5.53	5.42	0.06
QSort	2.78	2.64	2.68	2.69	0.05
QSort on Compressed	2.45	2.47	2.48	2.48	0.05
Andersson	3.63	3.60	3.67	3.61	0.04
Fast Sort	1.24	1.25	1.22	1.24	0.02
Fast Sort on Compressed	1.00	1.04	1.02	1.03	0.02
Binary Search	1.63	1.64	1.62	1.65	0.02
Binary Search on Compressed	1.36	1.38	1.37	1.38	0.02

5. CONCLUSIONS

In this work we study the benefits of order preserving compression for sorting strings in the word RAM model. First we propose a simple linear approximation algorithm for optimal order preserving compression, which acts reasonably well in comparison with optimum algorithms, both in theory and in practice. The approximation is within a constant additive term of both the optimum scheme and the information theoretical ideal, i.e. the entropy of the probabilistic distribution associated to the character frequency. We then test the benefits of this algorithm using the sorting algorithm of Andersson for the word-RAM as well as Bentley and Sedgewick's fast MKQSort. We present experimental data based on a 1GB web crawl, showing that Fast MKQSort and Andersson on compressed data are more efficient.

Acknowledgements We wish to thank Ian Munro for helpful discussions on this topic, as well as anonymous referees of an earlier version of this paper for their helpful comments.

REFERENCES

Andersson, A. 1994. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1996)*. 135–141.

Andersson, A., Hagerup, T., Nilsson, S., and Raman, R. 1995. Sorting in linear time? In *STOC: ACM Symposium on Theory of Computing (STOC)*.

Andersson, A. and Nilsson, S. 1994. A new efficient radix sort. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*.

Andersson, A. and Nilsson, S. 1998. Implementing radixsort. *ACM Journal of Experimental Algorithms 3*, 7.

Antoshenkov, G. 1997. Dictionary-based order-preserving string compression. *VLDB Journal: Very Large Data Bases 6,* 1 (Jan.), 26–39. Electronic edition.

Bayer, P. J. 1975. Improved bounds on the costs of optimal and balanced binary search trees. *Master's thesis. Massachussets Institute of Technology (MIT).*

Bell, T. C., Cleary, J. G., and Witten, I. H. 1990. *Text Compression.* Prentice Hall.

Bentley, J. L. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*. 360–369.

Farach, M. and Thorup, M. 1998. String matching in lempel-ziv compressed strings. *Algorithmica 20,* 4, 388–404.

Gilbert, E. N. and Moore, E. F. 1959. Variable-length binary encoding. *Bell Systems Technical Journal 38*, 933–968.

Hu, T. C. 1973. A new proof of the T-C algorithm. *SIAM Journal on Applied Mathematics 25,* 1 (July), 83–94.

Kärkkäinen, J. and Ukknonen, E. 1996. Lempel-ziv parsing and sublinear-size index structures for string matching. In *Proc. of the 3rd South American Workshop on String Processing (WSP '96).* 141–155.

Knuth, D. E. 1997. *The Art of Computer Programming : Fundamental Algorithms*, Third ed. Vol. 1. Addison–Wesley. ISBN: 0–201–89683–4.

Larmore, L. L. and Przytycka, T. M. 1998. The optimal alphabetic tree problem revisited. *Journal of Algorithms 28,* 1 (July), 1–20.

Moura, E., Navarro, G., and Ziviani, N. 1997. Indexing compressed text. In *Proc. of the 4th South American Workshop on String Processing (WSP'97).* Carleton University Press, 95–111.

Mumey, B. M. 1992. Some new results on constructing optimal alphabetic binary trees. *Master's thesis, University of British Columbia.*

Shannon, C. E. 1948. A mathematical theory of communication. *Bell Syst. Technical Jrnl. 27*, 379–423, 623–656.

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory, Vol.IT-24 5.*