PAPER

Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting

Kunihiko SADAKANE[†*], *Nonmember and* **Hiroshi IMAI**[†], *Regular Member*

SUMMARY Two new algorithms for improving the speed of the LZ77 compression are proposed. One is based on a new hashing algorithm named two-level hashing that enables fast longest match searching from a sliding dictionary, and the other uses suffix sorting. The former is suitable for small dictionaries and it significantly improves the speed of `gzip`, which uses a naive hashing algorithm. The latter is suitable for large dictionaries which improve compression ratio for large files. We also experiment on the compression ratio and the speed of block sorting compression, which uses suffix sorting in its compression algorithm. The results show that the LZ77 using the two-level hash is suitable for small dictionaries, the LZ77 using suffix sorting is good for large dictionaries when fast decompression speed and efficient use of memory are necessary, and block sorting is good for large dictionaries.

key words: *LZ77, hash,* `gzip`*, suffix sorting, block sorting*

1. Introduction

Many data compression schemes have been developed, and they are selected according to their compression speed, decompression speed, compression performance, memory requirements, etc. The LZ77 compression scheme [16] is a lossless compression scheme. Now it becomes a basis of many compression schemes. Its decompression speed is very fast and the memory required is small.

The LZ77 scheme compresses a string from left to right. It first finds a prefix of a string to be encoded from the string already encoded called *dictionary*. Then the prefix is encoded by its length and the distance between it and the string in the dictionary. The size of the dictionary is usually limited because of memory and compression time limitations, and therefore the dictionary stores only the newer part of the string. This type of dictionary is called a *sliding dictionary*.

To compress a string well, we have to find the longest match string in the dictionary. It is also important to find the nearest one among the longest match strings because the nearest one is encoded in fewer bits. The most time-consuming task in the LZ77 compression is to find the longest match strings. Hence the main topic of this paper is to find them quickly.

Manuscript received January 11, 2000.

Manuscript revised May 10, 2000.

[†]The authors are with the Department of Information Science, The University of Tokyo, Tokyo, 113-0033 Japan.

[*]Presently, with the Department of System Information Sciences, Graduate School of Information Sciences, Tohoku University.

Though the LZ77 has significant features described above, it is difficult to implement a fast encoder in practice. The LZ77 compression using the sliding dictionary can be done in linear time [10]. However, the algorithm requires huge memory and it is not fast in practice. Another problem is that it cannot find the nearest string in the dictionary. This causes compression loss.

The problems can be solved in part by using hashing algorithms. Almost all programs using the LZ77 scheme, for example `gzip`, `Info-ZIP`, `PKZIP`, `lha` and `arj`, use hashing data structures because of practical speed and memory efficiency [4]. Among them, `gzip`[6] is a typical and commonly used implementation of the LZ77 scheme. Though the hashing algorithms are fast enough for many strings, they become extremely slow for some strings. This is a reason to consider new algorithms for the LZ77.

There is another reason to improve the speed of the LZ77, especially `gzip`. It is used for comparisons with other compression algorithms in terms of their compression ratio and speed. Because compression speed depends on both the algorithm and its implementation, the speed of `gzip` needs to be improved for fair comparison between compression algorithms. For example, Balkenhol et al. [2] showed that their block sorting compression program is superior to `gzip` with `-9` option in both compression ratio and speed for the Canterbury corpus [1]. However, the reason why their algorithm is faster than `gzip` is that `gzip` becomes very slow for files E.coli and kennedy in the corpus. As will be shown in Sect. 5 of this paper, `gzip -9` is 13 times slower than our improved implementation of `gzip` for E.coli. In this regard, it is inappropriate to use the original program `gzip` as a representative program for fair comparison without reservation. Therefore it is important to improve speed of compression algorithms not only for practical reasons but also from academic interest.

In this paper, we consider increasing the speed of the LZ77 compression scheme. We propose two algorithms; one uses a new hashing technique called *two-level hashing* and the other uses suffix sorting. We improve the compression speed of `gzip` by the two-level hashing without sacrificing compression performance. This feature has not been achieved by the existing LZ77 programs. Their approach is to abandon finding the longest match strings. As a result, compression ratio will decrease. Our algorithm can be applied not only

gzip but also other LZ77 compression programs, which usually use hashing. By using the two-level hashing, the compression speed for English text files is also 30% to 60% faster and becomes three to five times faster for PostScript files of English documents created by dvi2ps. As mentioned above, it becomes 13 times faster than the gzip for the file E.coli. Because we only change the implementation of the function to find longest match strings, the decoder can be used as it is.

We also use a new development in text retrieval, the suffix array [12], to find the longest match strings. It is an array of lexicographically sorted indices of all suffixes of a string. Because the longest match string can be found by scanning a small part of the suffix array, finding it is faster than using hashing. Moreover, recently a fast suffix sorting algorithm was proposed [11].

The suffix array is a static data structure. Though dynamic tree structures such as binary trees also allow fast string searches, their construction is incremental and slow as is shown in [4]. On the other hand, the suffix array can be constructed by a batch procedure, which may be faster than dynamic data structures. Therefore we examine whether it is suitable for LZ77 compression to use suffix sorting. It is natural to compare the speed of LZ77 compression with block sorting compression [5], [14] because both schemes can be implemented by using suffix sorting. The block sorting compression became famous now for the reason of a good balance of compression speed and ratio. It is therefore important to test which of the two compression scheme is suitable for each size of the dictionary.

As a byproduct of the speed up, it becomes practically tractable to use a wider sliding dictionary. By using a wider dictionary, the compression ratio is improved [15]. We vary the dictionary size of the LZ77 and the block size in the block sorting, and compare their compression ratios and speeds for a collection of html files and articles of a newspaper. We find by experiments that the LZ77 compression using a suffix array is faster than hashing algorithms for very large dictionaries. We also find that the block sorting is superior to the LZ77 in terms of both compression ratio and speed.

2. Definitions and the Algorithm of gzip

First we define some notations.

- $X = x[1..N] = x_1 \dots x_N$: a string to be compressed where x_i is a character in an alphabet Σ
- $S_p = x[p..N]$: the p-th suffix of X
- $D_p = x[p - \text{DSIZ}..p-1]$: the sliding dictionary of size DSIZ when $x[1..p-1]$ has already been encoded

To encode the rest of the string $x[p..N]$, the LZ77 scheme finds the longest string $x[j..j+l-1]$ ($p-\text{DSIZ} \le j \le p-1$) that begins in the dictionary D_p and matches with a prefix $x[p..p+l-1]$ of the suffix S_p, then encodes the prefix by its length l and the distance $p-j$. We call the string $x[j..j+l-1]$ as the longest match string of S_p. Since the longest match string may appear more than once in the dictionary, we define a function to find it as follows.

- *longest_match*(p) = (l,q) where l is the length of the longest match string $x[r..r+l-1]$ ($\max\{1, p-\text{DSIZ}\} \le r < p$) and q is the smallest value of $p-r$ among the r attaining the maximum match length. If $len = 0$, q is undefined.

Note that the longest match string may exceed the right boundary of the dictionary, that is, the index of its tail $j+l-1$ may be more than $p-1$.

For two parameters M_1 and M_2, if $l < M_1$, the character x_p is encoded as a literal by Huffman, Shannon-Fano or arithmetic codes. The program gzip uses the Shannon-Fano code. If $l > M_2$, l is limited to M_2 because this limit makes implementation easy and the code tree small. In the implementation of gzip, DSIZ $= 2^{15}$, $\Sigma = \{0, 1, \dots, 255\}$, $M_1 = 3$ and $M_2 = 258$.

We give a detailed description of the algorithm for the function *longest_match*(p) in gzip because it is a basis of our two-level hashing algorithm. The program gzip uses a chaining hashing method. All suffixes S_i in the dictionary are inserted in a hash table of size HSIZ. All substrings of X are prefixes of S_i; therefore we store only indices i of S_i to the hash table. All suffixes that have the same hash value are inserted in a linked list, with the newest string at the top of the list. This is useful for LZ77 compression because we first find the newest match string that can be encoded in shorter bits than older ones. The linked lists in the hash table are represented by two arrays: **head** and **prev** (see Fig. 1). The arrays **head** and **prev** have sizes HSIZ and DSIZ respectively, and these are defined as

- **head**[h_1]: the index j of the newest suffix S_j in the dictionary that has hash value h_1
- **prev**[i & DMASK]: the index j of the suffix S_j previously inserted in a list just before S_i

where DMASK = DSIZ $-$ 1. The hash function $f_1(p)$

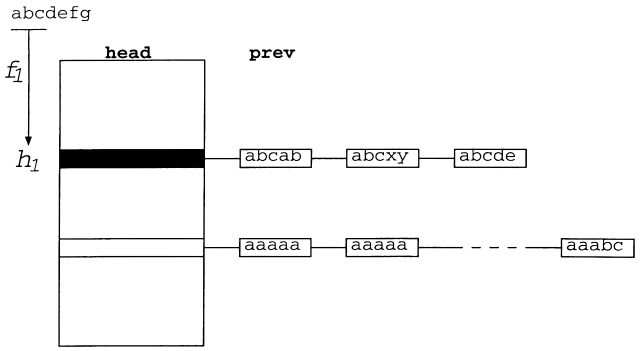

Fig. 1 Hash table of gzip.

is calculated from M_1 characters of the head of S_p:

$h_1 = 0$
for $(i = 0; i < M_1; i++)$ {
  $h_1 = (h_1 \texttt{<<} d)$ **xor** x_{p+i};
  $h_1 = h_1$ & HMASK;
}

where HMASK = HSIZ − 1. We assume that HSIZ is a power of two. This value can be incrementally updated from $f_1(p-1)$. Inserting or deleting suffixes is easy. Insertion is done as follows.

prev[p & DMASK] = **head**[h_1];

head[h_1] = p;

Deletion is unnecessary, since **prev** is a circular buffer of size DSIZ and $S_{p-\mathrm{DSIZ}}$ is automatically deleted by overwriting S_p. In `gzip`, HSIZ = DSIZ = 2^{15} and $d = 5$.

Calculating each hash value and inserting/deleting each suffix are quick, but searching is much slower. To search the longest match string of S_p, we first update the hash value $h_1 = f_1(p)$. Next we look up the hash table to get an index to the head of a list in the hash table. Then we compare S_p with each S_i in the list. It is enough to traverse only the list to find longest match string of length at least M_1.

3. Two-Level Hashing Algorithm

3.1 An Idea for Improving Speed

The hash value in `gzip` is calculated by only $M_1 = 3$ characters; hence many suffixes are inserted in the same list and searching becomes slower. For example, in some binary files, consecutive zeroes appear and their substrings have the same hash value. To improve the hash function, we should calculate the hash value by $m > M_1$ characters. However, if the longest match string in the list is shorter than m, we must then search other lists in the hash table that have shorter matched strings for better compression performance. Therefore we must change the hash value for searching all lists that may have the longest match string of length at least M_1. If the length of the longest match string is known, we can use an efficient hash function [13], but it is impossible to use such a hash function in our case.

We improve the hashing method used in `gzip` by using a two-level hash. Suffixes which have the same hash value are divided further by a secondary hash function. The secondary hash table is separated into many blocks. Each block corresponds to a primary hash value h_1 and it includes lists of suffixes corresponding the secondary hash values. The primary hash value is the base of a block in the secondary table and the secondary hash value is an offset in the block. Both hash values determine a hash chain. Because the hash function of `gzip` mostly works well, we use it as the primary hash function $f_1()$ of our two-level hashing algorithm.

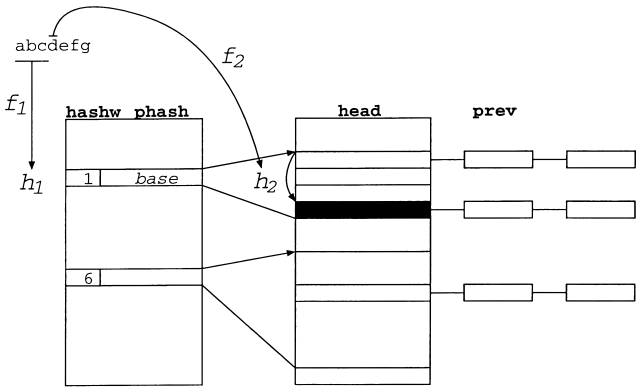

Fig. 2 Two-level hash table.

The secondary hash value h_2 is calculated from $x_{p+M_1+1}, x_{p+M_1+2}, \ldots$ The size of a block is determined by the number of suffixes having the same primary hash value.

3.2 Definitions

We define the two-level hash function. The primary hash function is the same as the function f_1 and the array **prev** is also used for representing hash chains.

- $h_1 = f_1(p)$: the primary hash value for a suffix S_p
- $h_2 = f_2(p)$: the secondary hash value for a suffix S_p
- $f_3(x_i) = x_i$ & 3: a function used to calculate $f_2(p)$
- $base$ = **phash**[h_1]: the base of a block in the secondary table corresponding to h_1
- b = **hashw**[h_1]: the number of characters used to calculate the secondary hash value corresponding to h_1
- **head**[$base + h_2$]: the index to the head of the list corresponding to h_1 and h_2

The sizes of tables **phash** and **hashw** are HSIZ. We set the size of the secondary hash table to DSIZ because the number of suffixes in the dictionary is DSIZ. We assume DSIZ = 2^w. Figure 2 shows the structure of the two-level hash table for the case $M_1 = 3$. The value h_1 is calculated from the first three characters 'abc' and h_2 is calculated from a character 'd' which follows the 'abc.'

3.3 Secondary Hash Function

The secondary hash value should be calculated quickly and incrementally like the primary hash value. We define the secondary hash value h_2 as a concatenation of $f_3(x_{p+M_1}), f_3(x_{p+M_1+1}), \ldots$. The idea of the secondary hash function comes from combinatorial hashing [9], where a hash value is the product of many smaller hash values. This kind of hash function is suitable for searching variable length strings because the number of lists

which have suffixes similar to S_p becomes small. On the other hand, the primary hash function does not fit searching variable length strings because the function really hashes suffixes in the dictionary. Therefore we use M_1 characters of suffixes to calculate the primary hash value.

The number of characters to calculate h_2 for S_p is $b = \mathbf{hashw}[h_1]$. Since the function f_3 takes two-bit values and the value h_2 is a concatenation of b pieces of the two-bit values, the maximum value of b becomes $\lfloor w/2 \rfloor$ and the maximum bit-length of a block in the secondary hash table becomes $w' = 2\lfloor w/2 \rfloor$. The secondary hash value for S_p is calculated by $x_{p+M_1}, x_{p+M_1+1}, \ldots, x_{p+M_1+b-1}$. Because $b \leq w'/2$, if we store the concatenation of $f_3(x_{p+M_1}), f_3(x_{p+M_1+1}), \ldots, f_3(x_{p+M_1+w'/2-1})$ as H_2, we can calculate h_2 by

$$h_2 = H_2 \gg (w' - 2 \cdot \mathbf{hashw}[h_1])$$

and we can update the value of H_2 for S_{p+1} from that of S_p by

$$H_2 = ((H_2 \ll 2) + f_3(x_{p+M_1+w'/2})) \ \& \ \mathrm{SHMASK}$$

where $\mathrm{SHMASK} = 2^{w'} - 1$. The initial value of H_2 becomes concatenation of $f_3(x_{1+M_1})$, $f_3(x_{2+M_1}), \ldots, f_3(x_{M_1+w'/2})$. Insertion becomes as follows.

$$base = \mathbf{phash}[h_1];$$
$$\mathbf{prev}[p \ \& \ \mathrm{DMASK}] = \mathbf{head}[base + h_2];$$
$$\mathbf{head}[base + h_2] = p;$$

Deletion is unnecessary, as in `gzip`, and updating the hash table is also the same as in `gzip`. We subtract DSIZ from each **head** and **prev** element.

3.4 The Sizes of Blocks in the Secondary Hash Table

The secondary hash table is divided into blocks. Each block corresponds to a primary hash value. If many suffixes have the same primary hash value, these should be stored in different lists in the secondary hash table. Therefore we determine the size of the block corresponding to a primary hash value h_1 to be proportional to the number of suffixes having the value h_1. We scan first DSIZ bytes of a file that is to be compressed. Then we determine $\mathbf{hashw}[h_1]$ for each primary hash value h_1 to be proportional to the logarithm of the frequency of h_1 and fill the secondary table with blocks, that is, $\mathbf{hashw}[h_1]$ becomes $\lfloor \lfloor \log_2 freq \rfloor / 2 \rfloor$ where *freq* is the frequency of h_1 in the first DSIZ bytes of a file. If $freq = 0$, we define $\mathbf{hashw}[h_1] = 0$ and the size of the secondary hash table becomes 1. In this case the secondary hash table for this value of h_1 is identical with the hash table of the original `gzip`. The values **phash** and **hashw** are computed as follows.

Count the frequency of each h_1 for S_1 to S_{DSIZ} in $\mathbf{tmp}[h_1]$.
$h = 0$;
for $(i = 0; i < \mathrm{DSIZ}; i++)$ {
    $\mathbf{phash}[i] = h$;
    if $(\mathbf{tmp}[i] > 0)$ {
        $b = \lfloor \lfloor \log_2 \mathbf{tmp}[i] \rfloor / 2 \rfloor$;
        $\mathbf{hashw}[i] = b$;
        $h += 2^{2b}$;
    } else $\mathbf{hashw}[i] = 0$;
}

The temporary array **tmp** is of size DSIZ and it is used for counting frequency of each h_1. If HSIZ $\geq$ DSIZ, we can use the **head** array instead of the **tmp** array so as not to increase the memory requirement. In `gzip`, DSIZ = HSIZ = 2^{15} and therefore growth of required memory is 96 Kbytes, 64 Kbytes for **phash** and 32 Kbytes for **hashw**.

3.5 Searching Strategy

When we encode a prefix of a suffix S_p, we first search the list in the secondary hash table that has the same primary and secondary hash values h_1 and h_2 with the suffix S_p. These hash values are calculated from the first M_1 characters and following $b = \mathbf{hashw}[h_1]$ characters of S_p. Therefore we need not search other lists that have the same primary hash value if we found prefixes of length at least $M_1 + b$ that match with S_p, otherwise we have to traverse the other lists to find prefixes of length l $(M_1 \leq l < M_1 + b)$.

To minimize the number of lists to traverse, we traverse the lists in decreasing order of the maximum match length between suffixes in the lists and S_p. We call this length as *maxlen*. To do so, we imaginarily divide the block corresponding to h_1 into groups. The group traversed first consists of only one list, $\mathbf{head}[\mathbf{phash}[h_1] + h_2]$. The *maxlen* of this group is M_2. The second group consists of three lists whose least significant two bits of the secondary hash value differ from h_2, that is, h_2 **xor** 1, h_2 **xor** 2 and h_2 **xor** 3. This means $maxlen = M_1 + b - 1$. We traverse these three lists to find matched strings longer than *bestlen* and shorter than $M_1 + b$ where *bestlen* is the length of the string matched so far. If we cannot find any matched string of length $M_1 + b - 1$, we next find a matched string of length $M_1 + b - 2$; therefore we change the secondary hash value to h_2 **xor** $4, \ldots, h_2$ **xor** 15 and search the 12 lists that correspond to these hash values, and so on. This operation continues until we either find a matched string that is *maxlen* long or finish searching all of the lists that correspond to the primary hash value. By using **xor** operation, we can easily change the order of traversing the lists for each suffix S_p. The algorithm is described as follows.

```
b = hashw[h1]; i = 0; j = 1; maxlen = M2;
for (k = 0; k ≤ b; k++) {
    for (; i < j; i++) {
        Find the longest match string of length
          up to maxlen from the list
          head[phash[h1] + (h2 xor i)].
    }
    If a matched string of length maxlen is found,
      then exit.
    j = j << 2; maxlen = M1 + b − k − 1;
}
```

Note that we must traverse all lists that have the same *maxlen* because we must find the closest string among the longest match strings. Assume that *bestlen* and *bestpos* are the length and position, respectively, of the longest matched string found so far. If we search for a matched string in a list that has not already been searched, we may find a matched string of length *bestlen* in a position that is newer than *bestpos*; therefore when the positions of suffixes S_i are newer than *bestpos* ($i >$ *bestpos*), we must find a matched string of length l that satisfies the conditions

$l \geq$ *bestlen* (if $i >$ *bestpos*) or
$l >$ *bestlen* (if $i <$ *bestpos*).

3.6 Reconstructing Hash Tables

The value of **hashw**$[h_1]$ is determined by the frequency of the value h_1 in the first DSIZ bytes of the text file to be compressed, but the balance of characters may change in the middle of the file, causing some long lists in the secondary hash table and slowing searching. If the length of a list in the table becomes too long, we reconstruct the hash tables. To do so, we first discard all tables except the dictionary, the raw string of the file. Next we count the frequency of the primary hash values and determine the size of blocks in the secondary hash table. Lastly we insert suffixes in the dictionary to the hash table. When to reconstruct a hash table is decided by experiments for the case DSIZ $= 2^{15}$. We reconstruct the hash table

- as soon as the length of a list is more than 2^{14}, or
- if the length of a list is more than 2^{2b+8} every time DSIZ characters are encoded.

4. Finding Longest Matches by Suffix Sorting

In this section we describe an algorithm for finding the longest match strings by using suffix sorting. All suffixes in a sliding dictionary are sorted in lexicographic order and their indices are stored in an array called the suffix array. In the suffix array, suffixes are arranged in order of match length with S_p. Therefore candidates of the longest match string with the suffix S_p is in the neighborhood of S_p. Therefore the number of suffixes compared with S_p decreases.

4.1 Definitions

The suffix array of a string $t[1..M]$ is an integer array $I[1..M]$. If $I[i] = j$ then the i-th suffix in lexicographic order is the suffix $t[j..M]$. An array $J[1..M]$ represents the lexicographical order of the suffixes. If $J[j] = i$ then the suffix $t[j..M]$ is lexicographically the i-th suffix. The array J is the inverse function of I, that is, $J[I[i]] = i$ for all i. A function $lcp(i, j)$ represents the length of the longest common prefix (lcp) of two suffixes $t[i..M]$ and $t[j..M]$.

4.2 Algorithm

We create the suffix array of a substring of length $(1+\alpha)$DSIZ containing the sliding dictionary D_p where α is a positive constant. When we search for the longest match strings of S_p, we skip suffixes which are outside of the current sliding dictionary $D_p = x[p-\text{DSIZ}..p-1]$. Every time αDSIZ characters were encoded, we create a new suffix array of the substring slided to the right by αDSIZ. That is, the last αDSIZ bytes of the substring are encoded by using preceding DSIZ bytes characters as the dictionary. Note that the first $(1+\alpha)$DSIZ characters of the string X are encoded without reconstructing the suffix array.

The following pseudo-code shows the function *longest_match*(p). First suffixes that are lexicographically smaller than S_p are traversed. Because the longest match strings are not arranged in lexicographic order, we must examine all suffixes that have maximum match length with S_p and find the closest one. The variable p_m represents the index of a suffix that matches p by l_m bytes. If a suffix has an index that is larger than p_m, p_m is updated. If the matched length between a suffix and S_p is less than the matched length found so far, the traverse is terminated. Next, suffixes that are lexicographically larger than S_p are traversed in the same way.

Note that p represents the global index ($1 \leq p \leq N$) of the suffix S_p, while q represents the local index ($1 \leq q \leq M = (1+\alpha)$DSIZ) of S_p in a substring $t[1..M]$ containing the sliding window D_p.

```
Calculate q from p.
i = J[q]; l_m = 0; i_1 = i − 1; i_2 = i + 1;
while (t[q] = t[I[i_1]]) {
    j = I[i_1];
    if (q − DSIZ ≤ j < q) {
        l = lcp(q, j);
        if (l > l_m or (l = l_m and j > p_m))
```

```
            {l_m = l; p_m = j;}
        if (l < l_m) break;
    }
    i_1 = i_1 − 1;
}
while (t[q] = t[I[i_2]]) {
    j = I[i_2];
    if (q − DSIZ ≤ j < q) {
        l = lcp(q, j);
        if (l > l_m or (l = l_m and j > p_m))
            {l_m = l; p_m = j;}
        if (l < l_m) break;
    }
    i_2 = i_2 + 1;
}
return (l_m, q − p_m)
```

The *lcp* function can be replaced by an array storing the length of lcp between adjacent suffixes in the suffix array, that is, $lcp(I[i-1], I[i])$ for all i in the suffix array. The value of $lcp(q, j)$ can be calculated by taking the minimum of $lcp(I[i-1], I[i])$ $(q < i \leq j \text{ or } j < i \leq q)$. Because the variables i_1 and i_2 are updated one by one, each lcp calculation can be done in constant time by using the lcp array.

5. Experimental Results

In this section we show several experimental results on compression speed and ratio. First we make a preliminarily experiment on relationship between dictionary size and time for finding the longest match strings using three algorithms: one-level hashing, two-level hashing and suffix sorting in Sect. 5.1. Then we compare our two-level hashing algorithm for DSIZ $= 2^{15}$ with `gzip` in Sect. 5.2. Then we make experiments on very large dictionaries to compare LZ77 comperssion by using two-level hashing and suffix sorting, and the block sorting compression in Sect. 5.3. We use a Sun Ultra60 workstation with 2048 MB memory except in Sect. 5.2.

5.1 Hashing Algorithms and Suffix Sorting

We compare time for finding the longest match strings for file 'bible.txt' in the Canterbury corpus [1] among the one-level hashing, the two-level hasing and suffix sorting. The dictionary size varies from 32 Kbytes to 1024 Kbytes. The algorithms using hashing use various sizes of HSIZ: 32 K, 64 K, 128 K, 256 K, 512 K and 1024 Kbytes.

In Fig. 3, *one-level hash* shows the time of the one-level hash algorithm, *two-level hash* shows that using our two-level hash algorithm, and *suffix sorting* shows that using suffix sorting for finding the longest match strings in the file 'bible.txt.' The x-axis represents the size of the sliding dictionary (DSIZ) and the y-axis represents the time. We set the parameter d in the primary hash function $f_1()$ as 5 if HSIZ is 32 K to 256 Kbytes and 6 if HSIZ is 512 K to 1024 Kbytes. For the other parameters, the same values as `gzip` uses. In the suffix sorting algorithm, $\alpha = 0.5$, that is, the size of a suffix array is 1.5DSIZ.

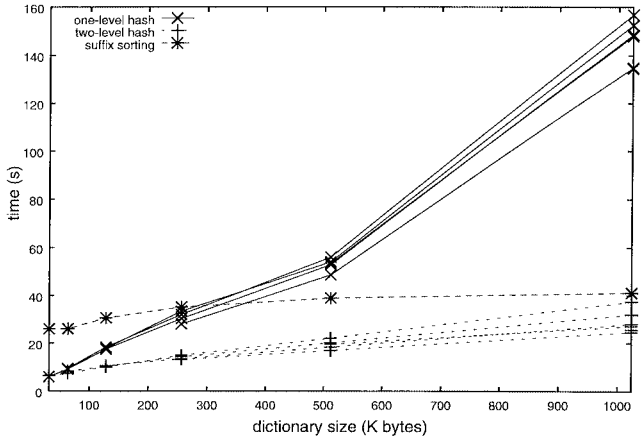

Fig. 3 Compression time and dictionary size.

Our two-level hashing algorithm is considerably faster than one-level and suffix sorting algorithms. The difference between the two-level hashing algorithm and the suffix sorting algorithm narrows as the size of the dictionary grows. Therefore, it is necessary to compare them for very large sliding dictionaries.

The results show that hashing algorithms become faster as the size of hash tables increases. Therefore we use hash tables that are the same size as the dictionary in the rest. We also exclude the one-level hashing algorithm from experiments because it is obviously slower than the two-level hashing.

5.2 The Two-Level Hashing and `gzip`

We tested compression speed and ratio of the original `gzip` and `gzip` using the two-level hasing. We used a Sun SPARC station 5 with 32 MB memory because the difference in compression time between the two programs becomes small if we use faster machines. We used files in the *Text Compression Corpus* [3], the *Canterbury Large Corpus* [1] and PostScript files of English documents [8] created by `dvi2ps`. We examined the time taken to compress the files and their size after compression. Compressed data are redirected to the null device of unix. Therefore compression time does not include time for writing compressed data to disk.

Table 1 shows compression time using "`gzip` -6," "`gzip` -9" and our two-level hashing algorithm for the Calgary and Canterbury corpora. First 14 files in the table come from the Calgary corpus and the rest 3 files come from the Canterbury large corpus. Among the

Table 1 Compression time (s) for corpora.

name	original size	`gzip` -6	`gzip` -9 (a)	ours (b)	(a/b)
bib	111261	0.64	0.83	0.64	1.29
book1	768771	6.85	8.55	4.79	1.78
book2	610856	4.11	5.11	3.41	1.49
geo	102400	1.69	2.60	2.14	1.21
news	377109	2.13	2.36	2.15	1.09
obj1	21504	0.10	0.17	0.20	0.85
obj2	246814	1.56	2.82	1.95	1.44
paper1	53161	0.29	0.33	0.28	1.17
paper2	82199	0.54	0.34	0.31	1.09
pic	513216	1.75	11.01	9.08	1.21
progc	39611	0.19	0.25	0.22	1.13
progl	71646	0.33	0.64	0.42	1.52
progp	49379	0.20	0.47	0.29	1.62
trans	93695	0.34	0.52	0.49	1.06
E.coli	4638690	87.52	386.30	29.07	13.28
bible.txt	4047392	30.86	59.71	26.12	2.28
world192.txt	2473400	13.14	18.67	15.64	1.19

Table 2 Compressed size of PostScript files in bytes.

name	size	`gzip` -6	`gzip` -9	ours
hasegawa	174370	49491	48879	48867
hayase	1430559	243790	236039	235810
ikeda	1295869	243986	235788	235503
kyoda	1247534	204960	199895	199643
masada	953292	177005	170480	170212
tanizo	791953	164867	158634	158496

Table 3 Compression time (s) for PostScript files.

name	`gzip` -6	`gzip` -9 (a)	ours (b)	(a/b)
hasegawa	0.7	3.6	0.7	5.14
hayase	4.5	32.8	8.1	4.04
ikeda	4.3	37.4	7.6	4.92
kyoda	3.3	25.1	8.0	3.13
masada	3.3	29.9	5.8	5.15
tanizo	2.8	23.4	4.4	5.31

Calgary corpus files, compression using our algorithm is 49% to 78% faster than "`gzip` -9" for English text files (book1, book2). For source lists of programs (progl, progp) it is between 50% and 60% faster. However our algorithm has no effect on small files (obj1, progc). For a file obj1, the speed becomes slower than "`gzip` -9." In these files most of the matched strings have a length of only three. In this case, our algorithm searches all of the elements with the same primary hash value and the speed of the search is not improved. The improved algorithm is faster for book1 and book2 and it is also faster than "`gzip` -6," but for the other files it is slower.

For the Canterbury large corpus files, our two-level hashing algorithm is also faster than "`gzip` -9." Furthermore, it is 13 times faster than "`gzip` -9" and three times faster than "gzip -6" for the file E.coli. The reason is as follows. The E.coli is a DNA sequence. Its alphabet size is four (a, t, g and c). Therefore strings in the sliding window are stored in only $4^3 = 64$ indices of the hash table in the `gzip` and many collision occur. On the other hand, by using the two-level hash algorithm the strings are divided in the secondary hash table. Note that the lowest two bits of the alphabet is 01 (a), 00 (t), 11 (g), and 11 (c), that is, characters 'g' and 'c' have the same secondary hash value $f_3()$. If we change the function $f_3()$ to have different values for the four characters, our algorithm runs much faster.

Table 2 and Table 3 show the compressed size and the compression speed for PostScript files. The compression speed is three to five times faster than "`gzip -9`." The compression ratio for PostScript files is slightly better than that of "`gzip` -9" because the length of a hash chain is limited to 4096 in `gzip` and `gzip` does not find the longest match string, whereas our algorithm searches all of the elements that may become the longest match string in a list. The compression ratio is of course better than "`gzip` -6." The program "`gzip` -6" traverses only the first 128 elements of each list and therefore cannot find the longest match string, whereas our two-level hash algorithm always finds the longest match string. In PostScript files, many long repetitions occur that are in the same hash chain. These cannot be distributed in a secondary hash table of the current size, which is why our algorithm is slower than "`gzip` -6" for these files.

5.3 Comparison between Hashing and Sorting

We show the test results for compression times and compression ratios of algorithms that are based on hashing and suffix sorting for various dictionary sizes. Note that compression time does not include encoding time; it includes only time spent searching the longest match strings. We also compare with another compression algorithm called *block sorting* [5] because it uses suffix sorting to compress a string. The block sorting compression consists of three steps: suffix sorting, move-to-front transformation, and Huffman or arithmetic encoding. The last step is omitted because we also omitted encoding step in the LZ77 compression.

5.3.1 Compression Time

Figure 4 shows the compression time for a collection of html files. Its size is about 90 Mbytes. Dictionary size varies from 32 Kbytes to 16 Mbytes. Because the files contain many long repetitions, by using large dictionaries compression ratio will be improved. We use five algorithms:

- *optlz77 (two-level hash)*: LZ77 using two-level hashing
- *optlz77 (sort+lcp+longest mach)*: LZ77 using suffix sorting and the lcp table
- *optlz77 (sort+longest match)*: LZ77 using suffix sorting
- *lz77 (sort only)*: LZ77 using suffix sorting which

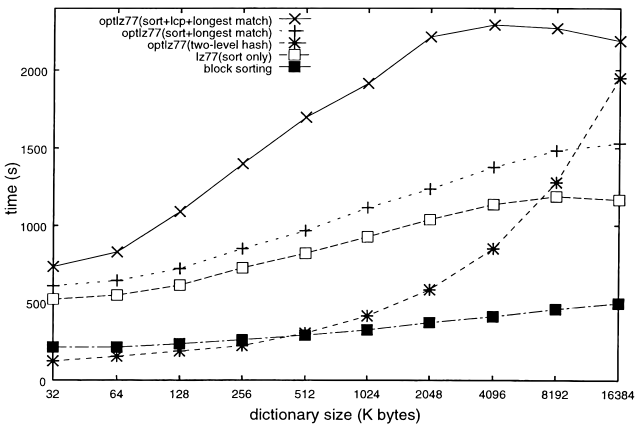

Fig. 4 Compression time and dictionary size.

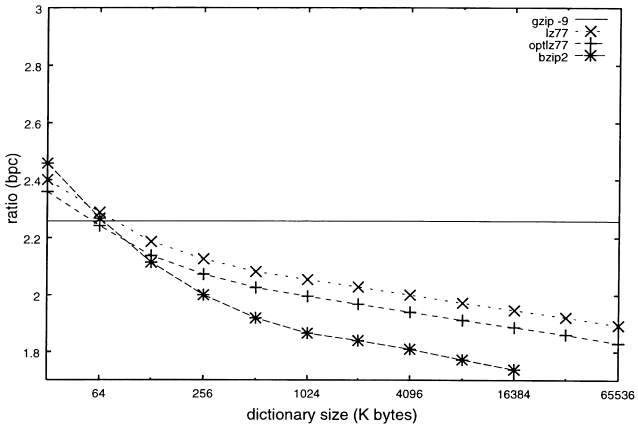

Fig. 5 Compression ratio for html files.

does not find the nearest longest match strings

- *block sorting*: the block sorting without encoding time.

The *optlz77 (sort+lcp+longest mach)* is slower than *optlz77 (sort+longest match)*. This implies that lcp calculation for all adjacent suffixes in a suffix array is not necessary. Note that this lcp computation can take place during suffix array construction (see Manber and Myers [12]), but this is not implemented in our programs.

The *block sorting* is about three times faster than the *lz77 (sort only)*. The reason is as follows. To find the longest match strings in LZ77 compression, a suffix array of size $(1+\alpha)$DSIZ is used and this is updated every αDSIZ bytes are compressed. A suffix array of size M is created in $O(M \log M)$ time. Therefore the total time becomes

$$O\left(\frac{N-\mathrm{DSIZ}}{\alpha\mathrm{DSIZ}} \cdot (1+\alpha)\mathrm{DSIZ}\log\{(1+\alpha)\mathrm{DSIZ})\}\right)$$
$$= O\left((N-\mathrm{DSIZ})\frac{1+\alpha}{\alpha} \cdot \log\{(1+\alpha)\mathrm{DSIZ})\}\right).$$

On the other hand, sorting time in the block sorting using blocks of size DSIZ is $O(N \log \mathrm{DSIZ})$ time. Therefore the block sorting is approximately three times faster than the LZ77 if $\alpha = 0.5$. For dictionaries smaller than 512 Kbytes, the LZ77 using two-level hashing is faster than the block sorting. If the size of dictionary is larger than 8192 Kbytes, the *optlz77 (sort+longest match)* algorithm is faster than the *optlz77 (two-level hash)* algorithm.

5.3.2 Compression Ratio

Here we compare compression ratio of LZ77 and block sorting for very large windows. `gzip` uses static codes, that is, it first stores output of the *longest_match* function in a buffer, then it calculates codes according to frequency of the output and encodes characters by using the codes. The compression ratio of `gzip` depends on the size of the buffer and it is difficult to find the optimal size. Therefore we use a simple compression scheme as follows:

- $(l, q) = longest\_match(p)$
- $(l_2, q_2) = longest\_match(p+1)$
- if $l < M_1$ or $l < l_2$, encode a character x_p by an adaptive arithmetic code.
- otherwise
  1. encode $l - M_1 + |\Sigma|$ by an adaptive arithmetic code
  2. encode bit-length of q by an adaptive arithmetic code
  3. encode q except the most significant bit

We use a technique called *lazy evaluation* to improve compression ratio. We encode a prefix of suffix S_p as a literal x_p if $l < l_2$. This technique is used in `gzip` and many other programs, and its effect is analyzed in [7]. It improves compression ratio by about 0.05 bits/character for both *optlz77* and *lz77* with various dictionary sizes in our experiments.

The adaptive arithmetic code encodes characters and match lengths according to their frequencies which are updated each time a character or a length is encoded. We call this program *optlz77* where *opt* means that this program finds the closest longest matches. Its performance will be close to the best program using the LZ77 scheme. We also use a program named *lz77*. Their difference is that *lz77* may not find the closest one among the longest match strings in the *longest_match* function. The function returns the first matched string among strings that match p. Therefore it may not find the closest longest match string and the compression ratio will decrease.

Because the *optlz77* and the *lz77* use the lazy evaluation, their compression speed is a little slower than the *lz77 (sort+longest match)* and the *lz77 (sort only)* in Fig. 4, respectively.

Figure 5 shows compression ratios of the *optlz77*, the *lz77* and `bzip2` for the html files. In the figures,

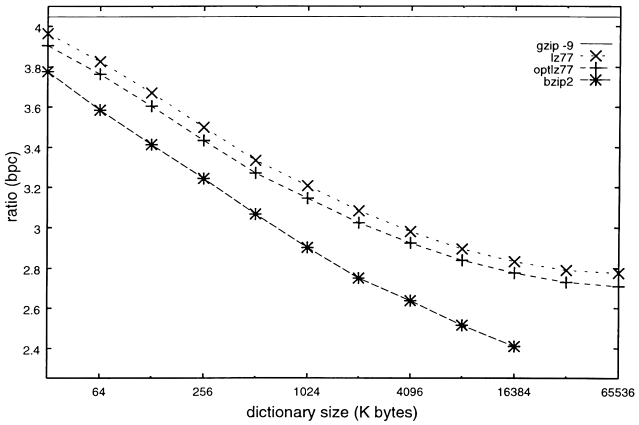

Fig. 6 Compression ratio for articles.

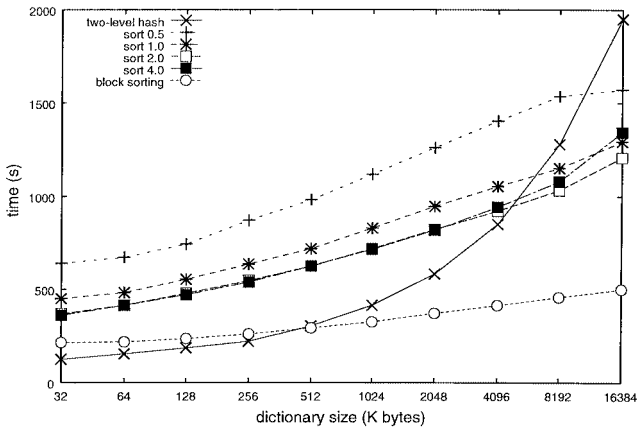

Fig. 7 Compression time and array size.

optlz77 and *lz77* show the compression ratio of LZ77 using suffix arrays, *gzip -9* shows that of `gzip`, and *bzip2* shows the compression ratio using `bzip2` version 0.1pl2, the block sorting compressor [14], which is modified to be able to use large blocks. The original `bzip2` uses a block of 100 Kbytes to 900 Kbytes in size, but here it is made between 32 Kbytes and 16 Mbytes with the least modification. The upper bound 16 Mbytes is tight for this modification. The program is also modified to use the fast suffix sorting algorithm [11].

The compression ratio of *optlz77* is better than that of *lz77*, which implies that finding the nearest longest match is important even if we use large dictionaries. The compression ratio of `gzip` is better than *optlz77* with a dictionary of 32 Kbytes because encoding of distances in `gzip` is optimized for the 32 Kbytes window.

The figure shows that the compression ratio of *bzip2* decreases faster than that of *optlz77* and *lz77* although the sliding window LZ77 is asymptotically optimal for all finite-alphabet stationary ergodic sources [15]. Therefore the block sorting has better compression ratio than the LZ77 for blocks of moderate sizes. The compression ratio of *optlz77* is better than *bzip2* if dictionary size is less than 128 Kbytes. In this case, *optlz77* is better than *bzip2* in both compression speed and ratio and the two-level hashing algorithm significantly improves compression speed.

About the compression ratios, a more remarkable result is obtained for all articles of Mainichi newspaper in 1995. The size of the text is about 100 Mbytes. Figure 6 shows compression ratios of the above algorithms. The compression ratios of our LZ77 implementations are reduced to about 70% by using a wider dictionary by a factor of 512. Even in this case, the *bzip2* is also superior to the *optlz77* and the *lz77*. Because both the html files and the articles are collections of similar kinds of texts and now we have many such kind of texts, the block sorting is useful to compress them.

5.4 Compression Time and Suffix Array Size

Figure 7 shows the compression time of the suffix sorting algorithm using suffix arrays of various sizes for the html files. The size of the suffix array is $(1+\alpha)$DSIZ. In the figure, *sort 0.5*, *sort 1.0*, *sort 2.0*, and *sort 4.0* show the compression time for $\alpha = 0.5, 1.0, 2.0, 4.0$, respectively. The number of suffixes to sort is $(N - \mathrm{DSIZ})(1+\alpha)/\alpha$ and becomes smaller as α increases. On the other hand, finding the nearest longest match string becomes slower by a factor of $(1+\alpha)$. If $\alpha = 4.0$, the compression time for large dictionaries is slower than when $\alpha = 2.0$. We find that an appropriate value of α is between 2.0 and 4.0.

5.5 Memory Requirements

Finally, we compare memory requirements. The two-level hash algorithm uses two integer array **phash** and **head** of size HSIZ, an integer array **prev** of size DSIZ, and an array **hashw** for small integers of size HSIZ. The suffix sorting algorithm uses two integer arrays of size $(1+\alpha)$DSIZ, I and J. These algorithms use almost the same amount of memory if HSIZ = DSIZ and $\alpha = 0.5$. The block sorting uses a suffix sorting algorithm which requires two integer arrays of size DSIZ. If $\alpha = 2.0$, the LZ77 using the suffix sorting algorithm becomes faster than when $\alpha = 0.5$. However, this requires 6DSIZ memory, which is three times larger than that required in the block sorting, and it is about twice as slow as the block sorting. Therefore the block sorting is superior to the LZ77 scheme in compression speed, compression ratio and required memory if the dictionary size is not too small. If we can use only limited memory or we want a fast decompression speed, then the LZ77 scheme with a small sliding dictionary is better, and the two-level hash algorithm is useful.

6. Conclusion

We considered increasing the speed of an LZ77-type data compression scheme. In the LZ77 compression, finding the longest match string is the most time-consuming process. Though the widely used `gzip` also uses the LZ77, its compression algorithm is not optimized. Therefore the compression speed of `gzip` needs to be improved, not only for practical purposes but also as it is the *touchstone* of compression algorithms. We proposed two algorithms for finding the longest match string. One uses two-level hash and the other suffix sorting.

We tested the compression time and ratio of the LZ77 method using hashing and suffix sorting, and the block sorting compression algorithm. The suffix sorting algorithm is faster than the two-level hash algorithm if the sliding dictionary is very large. To improve the compression ratio of the LZ77, it is important to find the nearest longest match string even if very large dictionaries are used. By comparing the compression time and ratios of the LZ77 and the block sorting, we concluded that:

- The suffix sorting algorithm can increase the speed of the LZ77 method if the dictionary is very large.
- However, the block sorting is superior to the LZ77 in compression speed and ratio and memory usage.
- For limited memory, the LZ77 is superior to block sorting and the proposed two-level hash algorithm significantly improves compression speed.
- It is well known that decoding of the LZ77 is faster and more memory efficient than the block sorting. By using the LZ77 with large dictionaries, we can improve compression ratio while preserving the features of the LZ77. In this case the suffix sorting algorithm is faster than the one-level and the two-level hash algorithm.

Acknowledgment

The work of the authors was supported in part by the Grant-in-Aid from the Ministry of Education, Science, Sports and Culture of Japan. The authors would like to thank the anonymous referees for their helpful comments.

References

[1] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," Proc. IEEE Data Compression Conference, pp.201–210, March 1997. `http://corpus.canterbury.ac.nz/`
[2] B. Balkenhol, S. Kurtz, and Y.M. Shtarkov. "Modification of the Burrows and Wheeler data compression algorithm," Proc. IEEE Data Compression Conference, pp.188–197, March 1999.
[3] T. Bell, J.G. Cleary, and I.H. Witten, Text Compression, Prentice Hall, 1990, `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/`
[4] T. Bell and D. Kulp, "Longest-match string searching for Ziv-Lempel compression," Software—Practice and Experience, vol.23, no.7, pp.757–771, July 1993.
[5] M. Burrows and D.J. Wheeler, "A block-sorting lossless data compression algorithms," Technical Report 124, Digital SRC Research Report, 1994.
[6] Jean-loup Gailly, `gzip`, 1993. `ftp://ftp.gnu.org/gnu/gzip/gzip-1.2.4.tar.gz`
[7] R.N. Horspool, "The effect of non-greedy parsing in Ziv-Lempel compression methods," Proc. IEEE Data Compression Conference, pp.302–311, 1995.
[8] Imai laboratory homepage. `http://www-imai.is.s.u-tokyo.ac.jp/`
[9] D.E. Knuth, The Art of Computer Programming, vol.3—Sorting and Searching, Addison-Wesley, Reading MA, 1973.
[10] N.J. Larsson, "Extended application of suffix trees to data compression," Proc. IEEE Data Compression Conference, pp.190–199, April 1996.
[11] N.J. Larsson and K. Sadakane, "Faster suffix sorting," Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999. `http://www.cs.lth.se/home/Jesper_Larsson/`
[12] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," SIAM J. Comput., vol.22, no.5, pp.935–948, Oct. 1993.
[13] P.K. Pearson, "Fast hashing of variable-length text strings," Communications of the Association for Computing Machinery, vol.33, no.6, pp.677–680, June 1990.
[14] J. Seward, `bzip2`, 1996. `http://www.muraroa.demon.co.uk/`
[15] A.D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," Proc. IEEE, vol.82, no.6, pp.872–877, June 1994.
[16] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Trans. Inf. Theory, vol.IT-23, no.3, pp.337–343, May 1977.

Kunihiko Sadakane received B.S., M.S., and Ph.D. degrees from Department of Information Science, University of Tokyo in 1995, 1997 and 2000, respectively. He is a research associate at Graduate School of Information Sciences, Tohoku University. He has won the Capocelli Award at the IEEE Data Compression Conference 1998. His research interests include text compression and databases.

Hiroshi Imai obtained B.Eng. in Mathematical Engineering, and M.Eng. and D.Eng. in Information Engineering, University of Tokyo in 1981, 1983 and 1986, respectively. In 1986–1990, he was an associate professor of Department of Computer Science and Communication Engineering, Kyushu University. Since 1990, he has been an associate professor at Department of Information Science, University of Tokyo.