LZ78 Compression in Low Main Memory Space[*]

Diego Arroyuelo[1], Rodrigo Cánovas[2], Gonzalo Navarro[3], and Rajeev Raman[4]

[1] Departamento de Informática, Universidad Técnica Federico Santa María, Vicuña Mackenna 3939, San Joaquín, Santiago, Chile, darroyue@inf.utfsm.cl
[2] LIRMM and IBC, 161 rue Ada, 34095 Montpellier Cedex 5, France, yigorc@gmail.com
[3] Deptartment of Computer Science, University of Chile, Beauchef 851, Santiago, Chile, gnavarro@dcc.uchile.cl
[4] Department of Informatics, University of Leicester, F33 Computer Science Building, University Road, Leicester, UK, r.raman@mcs.le.ac.uk

Abstract. We present the first algorithms that perform the LZ78 compression of a text of length n over alphabet $[1..\sigma]$, whose output is z integers, using only $O(z \lg \sigma)$ bits of main memory. The algorithms read the input text from disk in a single pass, and write the compressed output to disk. The text can also be decompressed within the same main memory usage, which is unprecedented too. The algorithms are based on hashing and, under some simplifying assumptions, run in $O(n)$ expected time. We experimentally verify that our algorithms use 2–9 times less time and/or space than previously implemented LZ78 compressors.

1 Introduction

The Ziv-Lempel algorithm of 1978 [19] (known as LZ78) is one of the most famous compression algorithms. Its variants (especially LZW [17]) are used in software like Unix's Compress and formats like GIF. Compared to the stronger LZ77 format [18], LZ78 has a more regular structure, which has made it the preferred choice for compressed sequence representations supporting optimal-time access [16] and compressed text indexes for pattern matching [7, 15, 3] and document retrieval [5, 6].

Compared to LZ77, the LZ78 compressed output is also easier to build. For example, a simple and classical implementation compresses a text of length n over an alphabet $[1..\sigma]$ into z integers, where $\sqrt{n} \leq z = O(n/\lg_\sigma n)$, in $O(n \lg \sigma)$ deterministic or $O(n)$ randomized time, using $O(z \lg n) = O(n \lg \sigma)$ bits of space. A comparable result for LZ77 was obtained only recently [11] and it required sophisticated compressed suffix array construction algorithms.

The time and main memory space required by compression algorithms is important. Building the compressed file within less main memory allows us compressing larger files without splitting them into chunks, yielding better compression in general. The fastest deterministic LZ78 compression algorithms require

[*] This collaboration started during the Dagstuhl Seminar 16431, "Computation over Compressed Structured Data". We also acknowledge the funding from Millennium Nucleus Information and Coordination in Networks ICM/FIC RC130003 (G.N.).

$O(n)$ time, but $O(n \lg n)$ bits of main memory [8]. If the main memory is limited to $O(n \lg \sigma)$ bits (*i.e.*, proportional to the input text size), then the time increases to $O(n \lg \lg \sigma)$ [11]. Finally, if we limit the main memory to $O(z \lg n)$ bits (*i.e.*, proportional to the compressed text size, like the classic scheme), then the compression time becomes $O(n(1 + \lg \sigma / \lg \lg n))$ [1], which improves the classic $O(n \lg \sigma)$ time. If we allow randomization, then the classic scheme yields $O(n)$ expected time and $O(z \lg n)$ bits of space.

In this paper we show that the LZ78 compression can be carried out within just $O(z \lg \sigma)$ bits of main memory, which is less than any other previous scheme, and asymptotically less than the size of the compressed file, $z(\lg z + \lg \sigma)$ bits. Ours are randomized and streaming algorithms. They read the text once from disk, and output the compressed file to disk as well, and therefore they may run within memory sizes unable to fit even the compressed file. One of our algorithms requires $O(n)$ expected compression time, but may rewrite the output multiple times, whereas the other takes $O(n \lg \sigma)$ expected time but writes the output only once. Both are able to decompress the file in a single $O(n)$-time pass on disk and using $O(z \lg \sigma)$ bits of main memory, where previous decompression algorithms need to store the whole compressed text in main memory.

Our results hold under some simplifying assumptions on randomness. Nevertheless, our experimental results demonstrate that these assumptions do not affect the practical competitiveness of the new algorithms, which outperform current alternatives in space and/or time by a factor from 2 to 9.

To obtain the result, we build on a hash-based trie representation [14], which has the advantage that the addresses of the nodes do not change as we insert new leaves, and that $O(\lg \sigma)$ bits are sufficient to encode the trie nodes since some of the information is implicit in their hash addresses. The main challenge is to design schemes so that the hash tables can grow as the LZ78 parsing progresses, so as to ensure that they have only $O(z)$ cells without knowing z in advance.

2 LZ78 Compression

The LZ78 compression algorithm [19] parses the text into a sequence of phrases. Assume we are compressing a text $T[1..n]$ and we have already processed $T[1..i-1]$ into r phrases $B_0 B_1 B_2 \cdots B_{r-1}$, where phrase B_0 represents the empty string. Then, to compute B_r we find the longest prefix $T[i..j-1]$ (with $j-1 < n$) that is equal to some B_q, with $q < r$. Then we define $B_r = B_q \,.\, T[j]$, which is represented as the pair $(q, T[j])$, and we continue the parsing from $T[j+1]$.

If we add a unique terminator character to T, then every phrase represents a different text substring. We call z the final number of phrases generated. It is known that $\sqrt{n} \le z = O(n / \lg_\sigma n)$, where σ is the size of the alphabet of T.

The usual way to carry out the parsing efficiently is to use the so-called LZTrie. This is a trie with one node per phrase, where the root node corresponds to B_0, and the node of $B_r = B_q \,.\, T[j]$ is the child of the node of phrase B_q with the edge labeled by $T[j]$. Since the set of phrases is prefix-closed, LZTrie has z nodes. Then, to process $T[i..j-1]$, we traverse LZTrie from the root downwards, following the characters $T[i], T[i+1] \ldots$ until falling out of the tree at a node representing phrase B_q. Then we create a new node for B_r, which is the child of B_q labelled by $T[j]$. Since the trie has z nodes, it requires $O(z(\lg n + \lg \sigma)) = O(n \lg \sigma)$ bits for the parsing, which can be done in $O(n \lg \sigma)$ deterministic time (using binary search on the children) or $O(n)$ randomized time (using hash tables to store the children, whose sizes double when needed).

The usual way to represent the LZ78 parsing in the compressed file consists of two (separate or interlaced) arrays, $S[1..z]$ of $z \lg \sigma$ bits, and $A[1..z]$ of $z \lg z$ bits. If $B_r = B_q \,.\, T[j]$ is a phrase, then we represent it by storing $A[r] = q$ and $S[r] = T[j]$. For decompressing a given phrase B_r, we follow the referencing chain using array A, obtaining the corresponding symbols from array S, until we read a 0 in A. Thus $S[r]$, $S[A[r]]$, $S[A[A[r]]$, $\ldots$ obtains B_r in reverse order in $O(|B_r|)$ time, and the complete text is decompressed in $O(n)$ time.

An alternative way to represent the LZ78 parsing [3] uses a succinct encoding of LZTrie, which uses just $2z + z \lg \sigma + o(z)$ bits, $2z + o(z)$ for the topology and $z \lg \sigma$ for the labels. It also stores an array $L[1..z]$, such that $L[r]$ stores the preorder number of the LZTrie node corresponding to phrase B_r. This array requires $z \lg z$ bits. To extract the text for phrase B_r, we start from the node with preorder $L[r]$ in LZTrie, and obtain the symbols labeling the upward path up to the root, by going successively to the parent in the trie. Using succinct tree representations that support going to the parent in constant time, this procedure also yields $O(n)$ total decompression time.

This second representation is more complex and uses slightly more space than the former, more precisely, the $2z + o(z)$ bits for the tree topology. Yet, it is sometimes preferred because it allows for operations other than just decompressing T. For instance, Sadakane and Grossi [16] show how to obtain any substring of length ℓ of T in optimal $O(\ell / \lg_\sigma n)$ time (*i.e.*, in time proportional to the number of machine words needed to store ℓ symbols). In this work, a different representation of LZTrie will allow us carrying out the compression within $O(z \lg \sigma)$ bits of main memory.

3 Previous Work on LZ78 Construction

A classic pointer-based implementation of LZTrie, with balanced binary trees to handle the children of each node, carries out the compression in $O(n \lg \sigma)$ time and $O(z \lg n)$ bits of space.

Jansson *et al.* [10] introduce a particular trie structure to represent LZTrie, which still uses $O(z \lg n)$ bits of space but reduces the construction time to $O\left(n \lg \sigma \cdot \frac{(\lg \lg n)^2}{\lg n \lg \lg \lg n}\right)$. The algorithm needs two passes on the text, each of which involves $n \lg \sigma$ bits of I/O if it is stored on disk.

Arroyuelo and Navarro [2] manage to perform a single pass over the text, in exchange for $2z \lg z$ additional bits of I/O, and a total time of $O(n(\lg \sigma + \lg \lg n))$. The peak memory usage is $z(\lg n + \lg \sigma + 2)$ bits. They use the compact LZTrie representation described in the previous section. An obstacle to further reducing the space is that they need to build the whole LZTrie before they can build

Reference	RAM space in bits	Compression time
Classic [19]	$O(z \lg n)$	$O(n \lg \sigma)$
	$O(z \lg n)$	$O(n)^*$
Fischer *et al.* [8]	$(1+\epsilon) n \lg n + O(n)$	$O(n/\epsilon^2)$
Köppl and Sadakane [11]	$O(n \lg \sigma)$	$O(n \lg \lg \sigma)$
Jansson *et al.* [10]	$O(z \lg n)$	$O\left(n \lg \sigma \cdot \frac{(\lg \lg n)^2}{\lg n \lg \lg \lg n}\right)$
Arroyuelo *et al.* [1]	$z(\lg n + \lg \sigma + 2)$	$O\left(n \lg \sigma \cdot \frac{1}{\lg \lg n}\right)$
Arroyuelo and Navarro [2]	$z(\lg n + \lg \sigma + 2)$	$O(n \lg \sigma + n \lg \lg n)$
This paper	$O(z \lg \sigma)$	$O(n)^*$

Table 1: Previous and new LZ78 compression algorithms. Times with a star mean expected time of randomized algorithms. We first list the classic schemes, then the deterministic methods, from fastest and most space-consuming to slowest and least space-consuming. At the end, our randomized method uses less space than all the others, and also matches the fastest ones in expectation.

the array L, because preorder numbers vary as new leaves are inserted. Later improvements on dynamic tries introduced by Arroyuelo *et al.* [1] reduce the time to $O\left(n \frac{\lg \sigma}{\lg \lg n}\right)$. Notice that this is $O(n)$ for small alphabets, $\sigma = \text{polylog}(n)$. However, the peak space usage remains the same.

Fischer *et al.* [8] finally obtained linear worst-case time. They construct the LZ78 parsing using $(1+\epsilon) n \lg n + O(n)$ bits of space in $O(n/\epsilon^2)$ time.

Recently, Köppl and Sadakane [11] showed how to construct the parsing in $O(n \lg \lg \sigma)$ time, using $O(n \lg \sigma)$ bits of working space; note this is $\Omega(z \lg n)$.

Table 1 shows all these previous results, and our contribution in context. Our results hold under some simplifying assumptions that are described in Section 8.

4 Dynamic Compact Tries

We will make use of the following data structure to maintain a dynamic trie of up to t nodes that uses $O(t \lg \sigma)$ bits, while supporting insertion of edges, and navigation upwards and downwards from nodes, within constant randomized time [14]. The structure has two components:

1. A closed hash table $H[0..M-1]$, where $M = m/\alpha$, m is an upper bound to the number of nodes, and the constant $\alpha < 1$ is the load factor to use. Table H is a simple array that stores information on the nodes of the trie, using only $\lg \sigma + O(1)$ bits per entry.
2. An array $D[0..M-1]$ to store information about the collisions (all entries initialized with value -1).

Each trie node y is identified with the position where it is stored in H; sometimes we will write $p(y)$ explicitly to refer to this position. This position will not change with time. The root is placed at an arbitrary position, say $H[0]$. Every other node y is represented by a pair (x, c), where x is (the position in H of) the parent of y and c is the character labeling the edge between x and y.

The hash function used to place y in H is $h(y) = ((a \cdot w(y)) \bmod P) \bmod M$, where a is an integer chosen at random in $[1, P-1]$, P is the first prime such that $P > M \cdot \sigma$, and $w(y) = p(x) \cdot \sigma + (c-1)$. The value we store in the cell of H associated with y is $v(y) = ((a \cdot w(y)) \bmod P) \operatorname{div} M$.

With this mechanism, since $P = O(M\sigma)$ [9, p. 343], it holds $v(y) = O(\sigma)$, and thus the values stored in H require $\lg \sigma + O(1)$ bits. With this information we can still reconstruct $(a \cdot w(y)) \bmod P = v(y) \cdot M + h(y)$, and then $w(y) = a^{-1} \cdot (a \cdot w(y)) \bmod P$, where $a^{-1} \bmod P$ is easily computed from a and stored with the index. From $w(y)$ we recover the pair (x, c), which allows us traversing the trie upwards.

On the other hand, to insert a new child $y = (x, c)$ from the position $p(x)$, we compute $h(y)$ and try to write $v(y)$ at $H[h(y)]$. If the cell is free (which we signal with $D[h(y)] = -1$), then we write $H[h(y)] \leftarrow v(y)$ and $D[h(y)] \leftarrow 0$. If the cell is not free, we probe consecutive positions $H[p]$ with $p = (h(y) + k) \bmod M$, for $k = 1, 2, \ldots$. The following cases may occur:

1. $D[p] = -1$, in which case we terminate with $H[p] \leftarrow v(y)$ and $D[p] \leftarrow k$, so that $D[p]$ indicates the number of probes between $h(y)$ and the final position p where y is finally written. Note that p will become $p(y)$, and from $p(y)$ we can recover $h(y)$ without knowing y, with $h(y) = (p(y) - D[p(y)]) \bmod M$.
2. $D[p] \neq -1$, $H[p] = v(y)$, and $(p - D[p]) \bmod M = h(y)$, thus node y is already stored in H, so we should not insert it.
3. $D[p] \neq -1$, but $H[p] \neq v(y)$ or $(p - D[p]) \bmod M \neq h(y)$, thus the cell is occupied by another node and we must continue with the next value of k.

Case 2 also shows how to traverse the trie downwards, from the current node towards its child labeled by c, to find the node $y = (x, c)$.

Note that the values stored in D are constant in expectation, as they record the insertion time for each element. Poyias et al. [14] show how D can be represented with a data structure using $O(z)$ bits and constant amortized-time operations. We refer the reader to their article for further details.

5 Using a Fixed Hash Table

In this section we show how to do the parsing of $T[1..n]$ within $O(n \lg \sigma / \lg_\sigma n)$ bits of main memory. This space is already $O(z \lg \sigma)$ on incompressible texts; we will later achieve it for all texts.

We use a compact dynamic trie to build the LZTrie associated with the LZ78 parse of T, and to compress T accordingly. We set m to an upper bound on the number of LZTrie nodes: m is the smallest number with $m(\lg_\sigma m - 3) \geq n$. Thus $m = \Theta(n / \lg_\sigma n)$. Further, we will use an array $L[0..z]$ to store in $L[r]$ the position in H where the LZTrie node of block B_r is stored. Each entry of L takes $\lceil \lg M \rceil = \lg n + O(1)$ bits, but the array is generated directly on disk.

To perform the parsing of a new phrase $T[i..j]$, we start from the trie root (say, x_0, with $p(x_0) = 0$), and use the mechanism described in the previous section to compute $x_1 = (x_0, T[i])$, $x_2 = (x_1, T[i+1])$, and so on until $x_{j-i+1} = (x_{j-i}, T[j])$ does not exist in the trie. At this point we insert $x_{j-i+1} = (x_{j-i}, T[j])$, write to disk the next value $L[r] \leftarrow p(x_{j-i+1})$, and continue with $T[j+1..n]$.

Overall, compression is carried out within the $O(n \lg \sigma / \lg_\sigma n)$ bits of main memory used by H and D, in $O(n)$ expected time if H is chosen from a universal family of hash functions, and T and L are read/written from/to disk in streaming mode. When we finish the parsing, we write H and D at the end of L in the file, and add some header information including n, σ, M, P, a, a^{-1}.

Decompression can also be made in streaming mode and using memory space only for H and D, which is not possible in classical schemes where each phrase is stored as a pointer to its earlier position in the file. We load the LZTrie into memory (*i.e.*, tables H and D). Now we read the consecutive entries of $L[1..z]$ in streaming mode. For each new entry $L[r] = p$, we start from $H[p]$ and decode $x_0 = (x_{-1}, c_{-1})$ from it; then we decode $x_{-1} = (x_{-2}, c_{-2})$, and so on, until we reach the root $x_{-s} = L[0]$. Then we append $c_{-s} c_{-s+1} \ldots c_{-2} c_{-1}$ to the decompressed text in streaming mode. The stack may require up to $z \lg \sigma$ bits in extreme cases, but this is still within our main memory budget. Its use can also be avoided in standard ways, at the expense of increased I/Os.

Note that this structure permits retrieving the contents of any individual block B_r, by traversing the LZTrie upwards from $L[r]$, just as done for decompression. This can make it useful as a succinct data structure as well.

The obvious disadvantage of this simple scheme is that it uses more than $O(z \lg \sigma)$ bits of space when T is highly compressible, $z = o(n/\lg_\sigma n)$ (that is, when it is most interesting to compress T!). A simple workaround is to start assuming that $z = O(\sqrt{n})$, since $\sqrt{n}$ is the smallest possible value for z. If, during the parsing, this limit is exceeded, we double the value of z and repeat the whole process. Since we may rerun the process $O(\lg z)$ times, the total expected time is $O(n \lg z) = O(n \lg n)$ (in LZ78, $\lg z = \Theta(\lg n)$). In exchange, the main memory space is now always $O(z \lg \sigma)$ bits. Further, the extra space added to the compressed file due to the tables H and D is just $O(z \lg \sigma)$. Apart from the increased time, a problem with this scheme is that it reads T several times from disk, and thus it is not a streaming algorithm. In the next sections we explore two faster solutions that in addition scan T only once.

6 Using a Growing Table

We can obtain $O(z \lg \sigma)$ bits of space for any text by letting H and D grow as more blocks are produced along the parsing. We start with a hash table of size $\sqrt{n}/\alpha^2$, since $\sqrt{n}$ is a lower bound on z. Then, whenever the load factor in H reaches α, we allocate a new table H' (and D') with size multiplied by $1/\alpha$, and load them with all the current trie nodes. We will read T only once, but we will still perform multiple rewriting passes on L.

The main challenge is how to remap all the nodes x from H to H', since their position $p(x)$ in H are their identity, which is mentioned not only in L but also in their children $y = (p(x), c)$. That is, in order to map y to its new position $p'(y)$ in H', we need to know the mapped position $p'(x)$ of its parent x, that is, we must map the LZTrie nodes top-down. Yet, we cannot simply perform a DFS traversal on LZTrie, because we cannot efficiently enumerate the children of a node x in less than $O(\sigma)$ time.

We remap the nodes as follows. We traverse L from left to right (on disk), and traverse upwards from each position $L[r]$ in H up to the root. All the nodes from the parent of $L[r]$ must already exist in H', so we stack the symbols traversed in the upward path on H and use them to traverse downwards in H' from the root. Then we insert in H' (and D') the new node that corresponds to $L[r]$, and rewrite $L[r]$ with the new position in H'. If B_r corresponds to $T[i..j]$, then our retraversal costs $O(j - i)$ time, so the time to retraverse $T[1..n']$ is $O(n)$. Since we perform $O(\lg z)$ passes, the total cost may reach $O(n \lg z) = O(n \lg n)$.

We can reduce the time to $O(z \lg_\sigma n) = O(n)$ by storing, when we have to load H', $O(z/\lg_\sigma n)$ sampled nodes of H in a (classic) hash table W, which stores the position in H' of each sampled position in H. Table W uses $O(z \lg \sigma)$ bits, which is within our budget. We will guarantee that every node of LZTrie whose depth is a multiple of $\lg_\sigma n$ and whose height is at least $\lg_\sigma n$ will be sampled. This ensures that $O(z/\lg_\sigma n)$ nodes are sampled and that we traverse less than $2\lg_\sigma n$ nodes of H from any cell $L[r]$ before reaching a sampled node, from which we can descend in H' and insert $L[r]$ in time $O(\lg_\sigma n)$. Thus we do the translation in $O(|L| \lg_\sigma n)$ time. Since the size of L grows by a factor of $1/\alpha$ each time we create a larger table, the total work amounts to $O(z \lg_\sigma n)$. To obtain the sampling invariant, we start by sampling the root. Then, every time we traverse from the node of $L[r]$ upwards, if we traverse $2\lg_\sigma n$ cells or more before finding a sampled node, we sample the node we traversed that is at distance $\lg_\sigma n$ from the sampled node we reached.

Once H' and D' are built, we continue with them and discard H and D. The peak space usage of the tables, when old and new ones are active, is $(1/\alpha^2 + 1/\alpha)z \lg \sigma + O(z) = O(z \lg \sigma)$ bits. Note that we can always keep the entries of L within $\lg z + O(1)$ bits, slightly expanding them when we retraverse L to rewrite the new positions in H'. At the end, L may need to point to a table H whose size is z/α^2, thus using $z \lg z + O(z)$ bits. To store H and D, we first write a bitvector B of length at most z/α^2 indicating which entries are $\neq -1$ in D. This requires $O(z)$ bits. Only the z filled entries of H and D are then written to the compressed file. The final compressed file size is then $z(\lg z + \lg \sigma) + O(z)$ bits.

Note that the $O(z)$ bits spent in L can be eliminated with a final pass on L replacing $L[r]$ by $rank_1(B, L[r])$, which is the number of 1s in B up to position $L[r]$. This can be computed in $O(z)$ time, and the values can be recovered in $O(z)$ time at decompression time using the complementary query $select_1(B, L[r])$ [4].

7 Using Multiple Hash Tables

A way to avoid rebuilding the hash table is to create additional hash tables apart from the original one, $H_0 = H$. When the load factor of H reaches α, we allocate a new table H_1, with $|H_1| = 2|H_0|$, where all the subsequent insertions will take place. When H_1 becomes full enough, we create H_2, with $|H_2| = 2|H_1|$, and so on, each time doubling the previously allocated space. Each table H_h has its own value M_h, prime P_h, and so on.

To properly address the nodes, we need to build a global address that can point to entries in any table. We regard the tables as their concatenation, that is, $H_0H_1H_2\ldots$ The addresses within table H_h are built by adding $|H_0H_1\ldots H_{h-1}|$ to the normal address computation within H_h. The prime P_h must then be larger than $(M_0+M_1+\ldots+M_h)\cdot\sigma$, so as to store any element $(p(x),c)$ where $p(x)$ is a global address. This requires only $O(1)$ extra bits per cell to store $P_h/M_h \le 2\sigma$.

Assume we are at a node x in a table H_g and want to add a child $y=(x,c)$ in the current table H_h. The entry $(p(x),a)$ will be inserted in H_h, leaving no indication in H_g of the existence of y. This means that, if we want to descend from x by c, we must probe tables $H_g, H_{g+1}, \ldots, H_h$ to see if it was inserted in later tables. Therefore, the cost of traversing towards a child grows to $O(\lg z)$, as we can build that many tables during the parsing. However, since the children are inserted later than their parents, the current table index does not decrease as we move down from the root towards the node where we will insert the new block, and thus we do these $O(\lg z)$ probes once per inserted block, for a total time of $O(z\lg z) = O(n\lg\sigma)$.

Instead, the parent x is decoded immediately from $y=(p(x),a)$, since $p(x)$ is a global address, and this allows decompressing in $O(n)$ time. Finding the table H_g from $p(x)$ is a matter of dividing $p(x)$ by $\sqrt{n}$ and then finding the logarithm in base 2, which is done in constant time in most architectures (and in theory, using constant precomputed tables of small size).

This technique has the advantage that it treats T and L in streaming mode, as it does not have to retraverse them. The values written on L are final (note that their width grows along the process, each time we start using a new table). These can be compacted as in the previous section if we are willing to perform a second pass on L.

8 Simplifying Assumptions

Our expected-case analysis inherits some simplifications from Poyias et al. [14], when it assumes constant expected time for hashing with linear probing.

A first one is that analyses usually assume that the hash function is chosen independently of the set of values to hash. In our scheme, however, the values $(p(x),c)$ to hash depend on the hash function $h(x)$ itself. So, at least in principle, the typical assumptions to prove 2-independence do not hold, even if we changed our function to the standard $((a_0 + a_1\cdot w(y)) \bmod M) \bmod P$ for randomly chosen a_0 and a_1.

File Name	Size n (Megabytes)	σ	Number z of phrases	Avg. phrase length (n/z)	Compr. ratio
`XML`	282.42	97	16,205,171	18.27	20.50%
`English`	1,024.00	237	96,986,744	11.07	37.96%
`Proteins`	1,129.20	27	147,482,019	8.03	48.55%
`Human Genome`	3,182.00	51	227,419,107	14.67	27.96%

Table 2: Text files used in the experiments.

Another issue is that 2-independence may not be sufficient to assume randomness in the case of linear probing. This has only been proved assuming 5-independence [12, 13]. To make it 5-independent, the component $a \cdot w(y)$ of our hash function should become $a_0 + a_1 w(y) + a_2 w(y)^2 + a_3 w(y)^3 + a_4 w(y)^4$. We do not know how to invert such a function in order to find $w(y)$ given $h(y)$ and $v(y)$.

In the next section we show, however, that those theoretical reservations do not have a significant impact on the practical performance of the scheme.

9 Experimental Results

In this section we experimentally evaluate our new algorithms with some previous implemented alternatives. We measure compression and decompression time, RAM usage and overhead of the final file size compared with the standard LZ78 format. All the experiments were performed on an Intel(R) Core(TM) i7-5500U CPU at 2.40 GHz. The operating system was Ubuntu 16.04.2 LTS, version 4.4.0-72-generic Linux kernel. Our compressors were implemented in C++11, using g++ version 4.8.4.

The texts considered are a highly compressible XML text, an English text, and a less compressible Protein file, all obtained from the Pizza&Chili Corpus[5]. We also used a DNA file generated by extracting a prefix of a human genome[6]. Table 2 lists the test files used and their main statistics. For the compression ratio we assume that each of the z phrases gives the parent phrase number and the symbol. For the former, the next 2^i phrases use $i + 1$ bits, starting from the second with $i = 0$. For the latter, we use $\lceil \lg \sigma \rceil$ bits.

Figure 1 shows the maximum RAM used by each structure during compression, and the resulting compression time. Our approaches are labeled HLZ (fixed hash table of maximum size, no rebuilding), MHLZ (multiple hash tables) and GHLZ (growing hash tables, no sampling). We obtain tradeoffs by using various load factors for the hash tables, $1/\alpha = 1.05, 1.10, 1.20, 1.40, 1.60$.

As previous work, we include LZ78-Min, the compact representation of Arroyuelo and Navarro [2], and LZ78-UC, their uncompressed baseline, both implemented in C.

[5] `http://pizzachili.dcc.uchile.cl/texts`

[6] `http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/est.fa.gz`

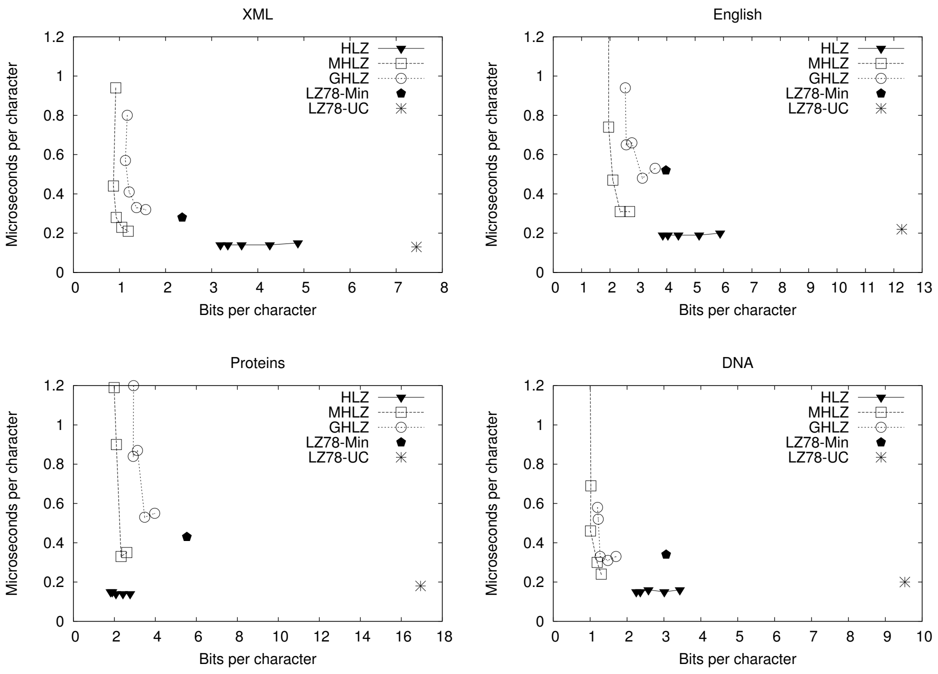

Fig. 1: Maximum RAM and time used during compression.

It can be seen that MHLZ always outperforms GHLZ in space/time, using 1.0–2.2 bits and 0.2–0.3 μsec per symbol with $1/\alpha = 1.40$. For the same space, the overhead of using multiple tables is lower than that of rebuilding the table, which implies rereading the L array from disk. In general, the time of MHLZ is very sensitive to high load factors, without significantly improving the space. With a sufficiently low load factor, instead, it outperforms all the others in time and space. It even gets close to the time of HLZ, always below 0.2 μsec, with much less space (with the exception of Proteins, where the final-size guess of HLZ is nearly optimal). The maximum space usage of GHLZ occurs when it has to expand the table, at which moment it has the old and new tables in RAM. This requires more space than MHLZ even when the MHLZ tables are emptier on average. LZ78-Min, instead, requires 2–3 times more space and is up to 4 times slower. Finally, LZ78-UC requires 6–9 times more space than MHLZ, and is not faster than HLZ.

Figure 2 shows the RAM used by each structure during decompression. This time GHLZ always obtains the best time of MHLZ but using slightly less space. GHLZ uses 0.9–1.8 bits and 0.1–0.2 μsec per symbol, even outperforming HLZ, which uses much more space (except on Proteins). GHLZ does not need to make the hash tables grow at decompression, thus it is much faster and uses less space

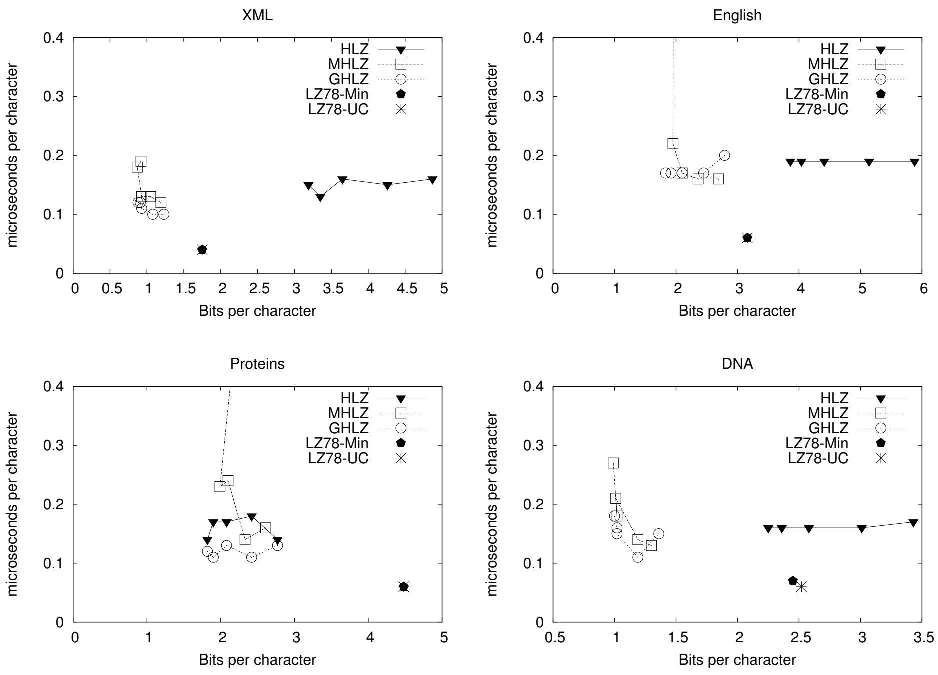

Fig. 2: Maximum RAM and time used during decompression.

than MHLZ, which has emptier tables. MHLZ is faster than for compression because it traverses the paths upwards, but it still uses multiple tables, and this poses some time overhead. LZ78-Min and LZ78-UC are identical for decompression, requiring 2–3 times more space but being 2–3 times faster than GHLZ.

Finally, Figure 3 shows the ratio between the actual compressed file size and the output of a classical LZ78 compressor (see Table 2). We exclude the HLZ baseline because it does not really compress. While MHLZ poses 30%–40% of overhead, GHLZ requires 25%–35%. We note that, to reach this overhead, we need to use $1/\alpha = 1.1$ or less, that is, almost the slowest. In this case, it is preferable to use GHLZ, which uses 1–3 bits and 0.5–0.8 μsec per symbol for compression and 0.1–0.2 μsec for decompression. If we want to have the fastest MHLZ compression times, we must accept an overhead of 40%–45%. On the other hand, LZ78-Min has an overhead of 4%–15%.

10 Conclusions

We have presented new LZ78 compression/decompression algorithms based on hashing, which under some simplifying assumptions use $O(z \lg \sigma)$ bits of main memory in expectation, while running in $O(n \lg \sigma)$ time for compression and

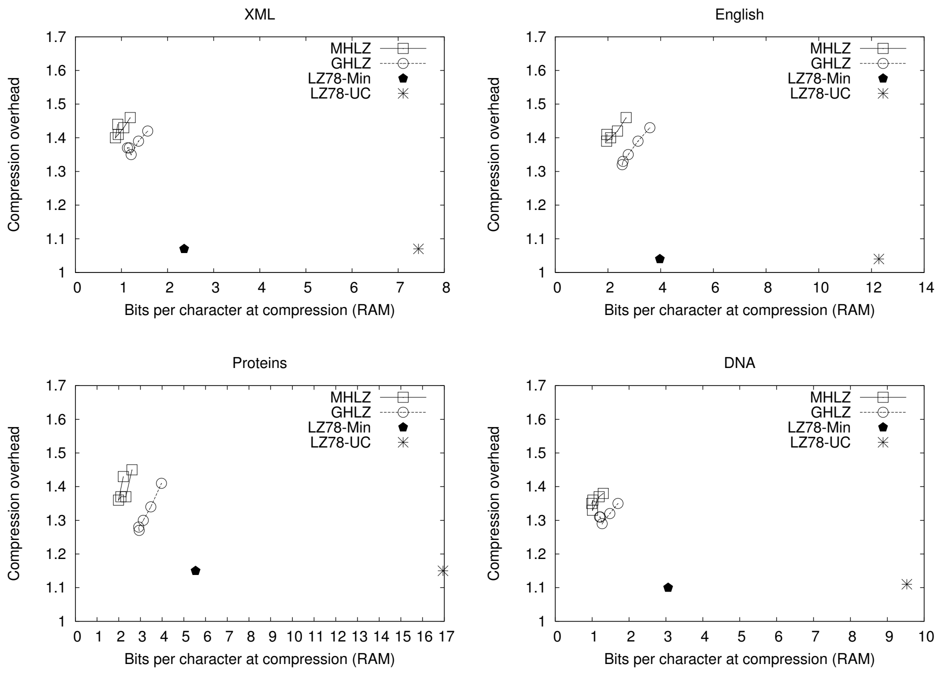

Fig. 3: Maximum RAM used at compression versus ratio of the final file size over the classical output size.

$O(n)$ time for decompression, where n is the text length, z the number of LZ78 phrases, and σ the alphabet size. Our algorithms read the text once, in streaming mode, and write the output to disk. There exists no previous algorithm using so little main memory.

Our experiments show that our new methods use 2–3 times less space for compression than the most space-efficient implemented compressor in the literature, while being up to 4 times faster. Compared to a classical baseline, our compressor uses 6–9 times less space and is only 50% slower. Our decompressor uses 2–3 times less space than both baselines, but it is 2–3 times slower.

For example, our compressor can use up to 3 bits and 0.8 μsec per symbol and our decompressor up to 2 bits and 0.2 μsec per symbol, posing a space overhead around 30% over the optimally compressed file.

Our compressors and the competing algorithms are publicly available at `https://github.com/rcanovas/Low-LZ78`.

An interesting line of future work is to use these hash-based tries as compressed text representations that retrieve any text substring [16], or for the compressed-space construction of LZ78-based text indexes [2].

Acknowledgements

We thank the reviewers for their insightful comments.

References

1. Arroyuelo, D., Davoodi, P., Satti, S.R.: Succinct dynamic cardinal trees. Algorithmica 74(2), 742–777 (2016)
2. Arroyuelo, D., Navarro, G.: Space-efficient construction of Lempel-Ziv compressed text indexes. Information and Computation 209(7), 1070–1102 (2011)
3. Arroyuelo, D., Navarro, G., Sadakane, K.: Stronger Lempel-Ziv based compressed text indexing. Algorithmica 62(1), 54–101 (2012)
4. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, University of Waterloo, Canada (1996)
5. Ferrada, H., Navarro, G.: A Lempel-Ziv compressed structure for document listing. In: Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 116–128. LNCS 8214 (2013)
6. Ferrada, H., Navarro, G.: Efficient compressed indexing for approximate top-k string retrieval. In: Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE). pp. 18–30. LNCS 8799 (2014)
7. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
8. Fischer, J., I, T., Köppl, D.: Lempel Ziv computation in small space (LZ-CISS). In: Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 172–184. LNCS 9133 (2015)
9. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers. Oxford University Press, 6th edn. (2008)
10. Jansson, J., Sadakane, K., Sung, W.: Linked dynamic tries with applications to LZ-compression in sublinear time and space. Algorithmica 71(4), 969–988 (2015)
11. Köppl, D., Sadakane, K.: Lempel-Ziv computation in compressed space (LZ-CICS). In: Proc. 26th Data Compression Conference. pp. 3–12 (2016)
12. Pagh, A., Pagh, R., Ruzic, M.: Linear probing with 5-wise independence. SIAM Review 53(3), 547–558 (2011)
13. Patrascu, M., Thorup, M.: On the k-independence required by linear probing and minwise independence. ACM Transactions on Algorithms 12(1), article 8 (2016)
14. Poyias, A., Puglisi, S.J., Raman, R.: m-Bonsai: a practical compact dynamic trie. CoRR abs/1704.05682 (2017), `http://arxiv.org/abs/1704.05682`, Preliminary version *Proc. SPIRE'15*, LNCS 9309.
15. Russo, L.M.S., Oliveira, A.L.: A compressed self-index using a Ziv-Lempel dictionary. Information Retrieval 11(4), 359–388 (2008)
16. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1230–1239 (2006)
17. Welch, T.A.: A technique for high performance data compression. IEEE Computer 17(6), 8–19 (1984)
18. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)
19. Ziv, J., Lempel, A.: Compression of individual sequences via variable length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)