

Longest match string searching for Ziv-Lempel compression

Timothy Bell and David Kulp

December 22, 1992

Department of Computer Science
University of Canterbury
Christchurch
New Zealand
phone: (+64 3) 3642 352
fax: (+64 3) 3642 999
e-mail: tim@cosc.canterbury.ac.nz

Summary

Ziv-Lempel coding is currently one of the more practical data compression schemes. It operates by replacing a substring of a text with a pointer to its longest previous occurrence in the input, for each coding step. Decoding a compressed file is very fast, but encoding involves searching at each coding step to find the longest match for the next few characters. This paper presents eight data structures that can be used to accelerate the searching, including adaptations of four methods normally used for exact match searching. The algorithms are evaluated analytically and empirically, indicating the trade-offs available between compression speed and memory consumption. Two of the algorithms are well-known methods of finding the longest match – the time-consuming linear search, and the storage-intensive trie (digital search tree.) The trie is adapted along the lines of a PATRICIA tree to operate economically. Hashing, binary search trees, splay trees, and the Boyer-Moore searching algorithm are traditionally used to search for exact matches, but we show how these can be adapted to find longest matches. In addition, two data structures specifically designed for the application are presented.

Introduction

Compression is becoming increasingly important as more and more information is stored on and transmitted between computers. The compression is *exact* if the original data can be recovered exactly from its compressed form. Exact compression is sometimes called *text compression* since it is most applicable to files of text, as opposed to non-textual files such as pictures, where some loss of quality is acceptable.

Many text compression techniques have been invented, with the best compression currently being achieved by complex modeling techniques [1, 2] linked to an arithmetic coder [3]. These methods can compress English text to about 2.2 bits per character (i.e. about 28% of the original size) [4]. The next best approach is a class of methods called Ziv-Lempel (LZ) coding, based on the work of Ziv and Lempel [5, 6]. Although LZ coding typically only achieves a compression of about 3 to 4 bits per character, it requires less memory and is faster than the modeling methods. Like the modeling techniques, LZ coders adapt to the type of text being compressed and can achieve good compression regardless of the language or subject of the input.

There are two main classes of LZ coding: those based on the LZ77 method [5] and the LZ78 method [6] respectively. Those derived from LZ77 are computationally intensive for encoding, but decoding is very efficient. LZ78 strikes a different balance with both encoding and decoding requiring a moderate amount of resources. Thus LZ78 is more appropriate for files that are not expected to be decoded often (for example, archives, backups and electronic mail) while LZ77 is the better method when a file is to be decoded many times, or is to be decoded on a smaller machine (for example, on-line manuals and news distributions). This paper explores techniques to accelerate the encoding for members of the LZ77 class of compressors. Many variations of LZ77 have been proposed, including LZSS [7, 8], LZH [9] and LZB [10]. All have the same problem of expensive encoding. The techniques explored in this paper are applicable to all of this family, although LZB was chosen to evaluate them as it usually gives better compression than the others.

Like all the LZ77-based methods, LZB employs a “sliding window” on the text. The window contains N symbols¹, with N typically being 8192. An example is shown in Figure 1 for $N = 4$, where the first nine symbols of a text have just been encoded. The window stores the 4 symbols most

¹for textual files, a symbol is usually one character

recently coded (“abba”) while a *lookahead buffer* stores the N symbols about to be coded (“babc”). The symbols in the text are numbered, from zero, in modulo $2N$, to enable them to be stored efficiently and indexed quickly in an array of $2N$ symbols operating as a circular buffer. At each coding step the compressor searches the window for the longest previous occurrence of the next few symbols. In the example the phrase “bab” has occurred previously starting at position 7. Note that although the matched phrase must begin in the window, it may extend into the lookahead buffer, but the match is limited to at most N symbols. The symbols matched are then replaced with a pointer (7,3), which indicates that the next few symbols can be located at position 7 in the window, and there are 3 of them. The window is then moved to the right by three symbols simply by overwriting symbols 5, 6, and 7 with the incoming symbols (“x,” “y,” and “z,”) and changing the start of the lookahead buffer to be position 4.

The combined window and lookahead buffer are of length $2N$, but the maximum allowed match size is restricted to M ; typically, $M = 128$ gives good compression.

The decoder maintains a window of N symbols, identical to the encoder’s window. When a pointer is received it simply copies the symbols referenced into the next positions in the window. Note that there is no problem if the pointer is recursive, since a symbol will be copied into the window before it is required.

The members of the LZ77 family of compressors differ mainly in the details of how they store a pointer, how they deal with new symbols that cannot be coded as pointers, and how they prime the window at the beginning of coding. Their common problem is to search the window for a longest match when N is large, and it is on this that we concentrate.

Searching algorithms

In the following descriptions of data structures the *input alphabet* is the set of q possible symbols that might be encountered in the input file. For example, for ASCII files $q = 128$, while for a byte-oriented file $q = 256$. A *phrase* or *substring* is a sequence of zero or more symbols.

The simplest method of searching the window is a linear search, which compares the lookahead buffer with each of the N positions in the window and selects the maximum match. Linear searching in the worst case is performed in $O(NM)$ time where M is the size of the maximum pattern

to be matched in the lookahead buffer. In most cases, however, linear searching is much faster than this and can be completed in $O(M + N)$ time.

Knuth, Morris, and Pratt [11] have designed an improved linear searching variant, KMP, which ensures that worst case behaviour is $O(M + N)$. The KMP algorithm pre-processes the search key and identifies repetitions within the key. For each position, i , in the key, KMP establishes a pointer to a previous position, j , in the key where comparisons should resume if comparisons fail at i . This allows the scanning algorithm to jump ahead through the source text by increments greater than one. Experimental results show that KMP is no better than the simple linear search for texts with little repetition within the pattern, such as English text, but KMP performs much better than the simple linear search for text such as pictures which contain many repeated symbols. For a maximum match size of M , the KMP algorithm requires M pointers to store the jumps for each position in the key. Generally, KMP shows little, if any, improvement over linear searching and is not useful, but the concept of preprocessing a search key for internal repetitions was used by Boyer and Moore to design a more powerful scanning algorithm.

Boyer and Moore [12] designed a variant to KMP that outperforms all other general scanning algorithms, i.e., algorithms that do not preprocess the source text to be searched. The Boyer-Moore algorithm scans the window from left to right, but compares symbols from right to left. When a mismatch occurs during comparison, the key shifts the exact number of steps to line up the symbol in the window with its occurrence in the key, and comparisons are then resumed. If a symbol encountered in the window does not occur in the key, then the key can be shifted beyond the mismatched symbol entirely. This requires a data structure to map each symbol in the alphabet to the position of the rightmost occurrence of that symbol in the key. The algorithm also includes a data structure similar to that required for KMP to identify any repetitions within the search key. The Boyer-Moore algorithm can be modified to locate the largest prefix of a string by scanning the window from right to left and comparing symbols from left to right. The data structures must be “reversed” as well [13].

Boyer-Moore guarantees linear performance in the worst case, and on average the algorithm searches in sub-linear time. This holds true for searches of exact matches or longest prefix matches. For an input alphabet of q symbols, q pointers are required to store the position of the leftmost occurrence of each symbol in the alphabet. Boyer-Moore also

requires M pointers for identifying self-repetition, as with KMP.

For both KMP and Boyer-Moore, searching requires some small overhead to pre-process the pattern before searching. Therefore, these algorithms perform better for large values of N and small values of M . If M is too small, however, then the advantages of the algorithms are lost; if M is very large, then performance may become impaired as the time required to pre-process the pattern dominates the search time.

A different approach involves developing data structures to index the source text in the window and allow matches to be identified rapidly. As symbols enter and leave the window the data structure is updated to account for the new selection of potential match positions available. Thus three operations are performed on the data structure: *insert* (a symbol entering the window – a potential starting point for a match), *delete* (a symbol leaving the window), and *search* (find the longest match for the lookahead buffer.) Usually the task of finding a match for the lookahead buffer can be combined with inserting it in the data structure.

Linked list

In this data structure a linked list is stored for each symbol in the input alphabet, and indicates all of the positions where the symbol can be found in the window. To find the longest match the substring starting at each position in the appropriate list is matched against the lookahead buffer, and the longest is selected. The list is maintained as a queue; as a symbol enters the window it is inserted at the end of the linked list, and as it leaves it is deleted from the head of the list. By storing a pointer to the head and tail of each list, the insert and delete operations can be performed in $O(1)$ time. The worst case for searching occurs when all the symbols in the text are the same. In this case all N window positions will be checked, and a match of length M will be found at each, requiring $O(NM)$ time. However, for most texts only a small proportion of the N entries need to be checked and the matches are usually just a few symbols long. For an input alphabet of q symbols, $2q$ pointers are required to store the head and tail pointers, and an additional N pointers are required to form the linked lists. The N pointers can be drawn from a continuous array, with the pointer at position i corresponding to the symbol at position i . Thus there is no need to store the symbol position in the linked list, since it can be determined from the location of its pointer.

A modification to the data structure just described is to store a list for every *pair* of symbols in the alphabet. The list corresponding to the first two symbols in the lookahead buffer is searched for a match. If no match of two or longer exists then the list will be empty, and the other list, indexing the most recent occurrence of the first symbol, should be checked to see if a match of length one can be found. The time complexity of this method is the same as for the single symbol lists, although in practice the lists will be shorter and the searching faster. More memory is required as there are now q^2 head and tail pointers plus another q pointers to the most recent occurrence of each single symbol, but only N link pointers are required as before.

The linked list method could be extended to use the first k symbols, but it rapidly becomes impractical because $O(q^k)$ storage is required.

Trie

A trie (also called a digital search tree) is a multiway tree that has a path from the root to a leaf for each string indexed. Figure 2 shows a trie that corresponds to the window in Figure 1. Each of the N positions in the window is treated as the start of a string of length M , and each of these strings is indexed by the trie. Each node identifies where the string can be found in the window. For example, to find the longest match for the lookahead buffer “babc”, the path “bab” is followed down the trie, at which point progress is blocked. The last node encountered indicates that “bab” can be found at position 7 in the window.

The type of trie shown in Figure 2 is impractical because each path to a leaf contains M nodes, requiring an excessive amount of storage for large values of M . However, this can be overcome by coalescing chains of nodes with single children, so one arc may represent several symbols (Figure 3). The modified structure is essentially a PATRICIA tree [14]. The symbols that label the arcs are not stored explicitly, but each node stores a pointer to where they are in the window, from which they can be obtained. For example, the label for the arc into the node labelled $(0, 2)$ can be obtained by looking at the two symbols starting at position 0 in the window.

To find a longest match a search is performed down the tree until the path is blocked by a mismatched symbol. The location of the match can be determined directly from the pointer in the last node consulted. To insert a new phrase the algorithm follows the corresponding path down the tree until it is blocked. If that point is on an arc between two nodes

then a new node is inserted on the arc. The new node corresponds to where the mismatch occurred, and a second new node is added as its child. If the mismatch occurs at a node then a new child node is simply added to it. If a leaf is reached then it should be replaced with a reference to the new (identical) substring. To delete a phrase from the trie the leaf that corresponds to the phrase is located and removed. If the leaf's parent now had only one child, the parent is also removed, being replaced with an arc from its parent to the only child. For a detailed example of this type of trie see McCreight [15], pp 263-265.

The time taken to search this modified trie is proportional to the length of the match, which in the worst case is $O(M)$, although typically it is just a few symbols. Every phrase in the source must be inserted, and insertion requires a search through the trie to find the location for the new phrase. Insertion, therefore, is more time-consuming over searching alone. In practice the children of a node are stored as a linked list of siblings, containing up to q nodes and introducing at worst a factor of q to the time. Each node stores two integers to point into the window, and two other pointers, one to the node's first child, and one to its next sibling. For each of the N substrings indexed by the trie, one leaf node is required, and possibly one internal node. Thus at most $2N$ nodes are required, giving a total of $4N$ integers and $4N$ pointers.

A related structure, called a suffix tree [15], inserts strings more efficiently by maintaining extra pointers within the tree, although it is unable to accommodate deletions easily. The suffix tree has been applied to a form of LZ77 coding labelled LZR [16] but because deletion is difficult it is avoided by discarding the tree when it is full and starting on another tree, which has been primed with recently coded symbols. The inefficiency of deletion makes the suffix tree impractical for a sliding window, but is of value if the window is allowed to grow continuously, as for LZR.

Hash table

There are several ways to use a hash table to find longest matches. One method [17] is to store all substrings in the window up to some length k into the hash table. To find the longest match for a phrase, look up its first symbol in the hash table; if that is found look up the phrase consisting of the first two symbols, and so on, until the probe into the table fails. If a match of length k succeeds then a longer match must be sought by other means. Brent [9] uses a variation of this that stores all substrings of length $1, 2, \dots$, until the substring being stored is unique.

The problem with both of these methods is that the number of substrings stored in the hash table is potentially very large.

A new approach evaluated here, similar to the linked lists described above, stores long matches economically by storing substrings of length 1, 2, 4 and 8 only. Separate chaining is used, so each hash table entry is a linked list of the positions where the corresponding substring can be found. Due to collisions, other positions may also be in the list and should be ignored during searching. To find the longest match for the lookahead buffer, the first symbol is hashed and the corresponding list is searched; if that is successful, then the substring comprising the first two symbols, then the first four, and so on is used. As soon as a match fails the algorithm reverts to the next smaller list and takes the longest match from that. If none failed, the length 8 list is searched. The maximum length of 8 was chosen because matches are typically about 4 symbols long, although matches of around 16 symbols do occur in English text.

The hash function chosen for the string of symbols $c_1c_2c_3\dots$ is

$$\begin{aligned} h_0 &= 0 , \\ h_{i+1} &= 4h_i + ord(c_i) . \end{aligned}$$

The hash table was chosen to have approximately 50% occupancy. Neglecting collisions, the appropriate list will be found in 4 or less probes. Searching the lists can be made reasonably efficient as follows. Suppose a suitable string was found in the 2's list, but not in the 4's. Then the best possible match must be 2 or 3 symbols. We continue to search the 2's list, and once a match of length 3 has been found in the the list the search can stop because no longer match is possible. An extreme case of this is the 1's list, where only the first entry will ever be used. Wasted storage can be avoided by keeping the 1's list separately as an array of q integers recording the most recent occurrence of each symbol in the window. A tail pointer is stored for each list in the hash table to allow deletion to be performed in $O(1)$ time.

The hash table has $2(6N - 1)$ entries — 3 pairs of entries (head and tail) for each of the N phrases at 50% occupancy, with 1 subtracted to obtain an odd (and probably prime) size. Each entry in the hash table is a pointer, and an additional $3N$ pointers are required for the linked lists (separate chains) giving a total storage requirement of $15N - 1$ pointers and q integers.

Binary search tree

Consider a binary search tree in which a number of strings have been inserted. It can be shown [8] that while inserting a new string into the tree, the path followed will encounter the string in the tree with the longest prefix in common with the string being inserted. Thus a binary tree can be used to find a longest match. For an LZ encoder, a binary search tree of N nodes is maintained, one node for each position in the window. The string used to compare a node with others is the N symbols starting at the corresponding position in the window. For example, Figure 4 shows a binary tree for the window in Figure 1. Each node represents one of the four substrings of length four that begin in the window. In practice a node need only store the position in the window that it corresponds to, since the string can be obtained from the window. To find the longest match for the lookahead buffer “babc”, the algorithm follows the path to insert the phrase in the tree (through “abba,” “bbab” and “baba,”) and notes the length of the match with each of the three strings encountered. When a leaf is reached the longest of the matches is chosen, which is the first three symbols of “baba”, at position 7 in the window. As symbols enter the window their corresponding strings are inserted in the tree using the normal algorithm for a binary search tree, and as each symbol leaves a node is deleted.

If the tree is reasonably balanced (as is the case if the input is English text) then each insertion, deletion and search will require string comparisons at $O(\log N)$ levels. In the worst case the tree will degenerate to a linked list requiring string comparisons at $O(N)$ levels or $O(NM)$ symbol comparisons for each update. Worst case behaviour occurs when a long run of one symbol is followed by a different symbol. For example, the phrase “aaabaaa” generates the substrings “aaab”, “aaba”, “abaa” and “baaa”. If these are inserted in the order they appear then degenerate behaviour results. This behaviour can be ameliorated by scrambling the order in which substrings are inserted.

The binary tree requires N nodes, which can be drawn from an array with node i corresponding to position i in the window [8]. Thus i can be determined from the node’s location in the array, and need not be stored explicitly in the node. Each node simply stores a pointer to its left and right children, and to expedite deletion, to its parents. The total memory requirement is $3N$ pointers.

Splay tree

A splay tree [18] is a self-adjusting binary search tree that attempts to maintain balance by rearranging the tree around a node after it is accessed. It is effective in that the amortized time to search a tree is $O(\log N)$. Since during compression we are usually only concerned with the overall time taken, and not individual access times, amortizing the speed is appropriate. Because splaying maintains the properties of the binary search tree, a longest match can be found using exactly the same method described for a binary search tree. The advantage is that the tree will automatically balance itself giving a worst case amortised time of $O(\log N)$. The splay tree has no extra memory requirements over an ordinary tree, and so can be stored as $3N$ pointers.

Empirical evaluation

The analyses of the searching methods are difficult to compare because they depend on different quantities; the linked list depends on how often a symbol occurs in the window; the trie on how long the common prefixes of strings are; and the binary tree on the order in which strings are encountered. To compare the methods they need to be evaluated with sample texts, and the results of such experiments are reported in this section. The eight search methods described in the previous section have all been used to implement an LZB coder, and are summarized in Table 1. The search routines were all written in the “C” programming language and executed on a Sun Sparcstation model 4/75 running SunOS 4.1.1. They were coded with speed in mind, and compiled with the highest level of optimisation available.

The performance of a method depends on the type of text being compressed. The worst case times for most of the methods occurs when there is a lot of order in the text, which is usually when the best compression is achieved. To obtain a contrast between the average case and worst case, two test files were used. The file “book” is a transcript of Thomas Hardy’s book *Far from the Madding Crowd*, and contains 768,771 characters. The file “picture” is a black-and-white bit-mapped picture of a book page, and is 513,216 bytes long. It contains a great deal of repetition where large amounts of white space are coded as runs of binary 0’s – for example, it ends with 36,316 consecutive 0’s. The “picture” file was chosen because it brings out the worst case behaviour of almost all of the searching methods.

Figure 5 shows the compression that LZB achieves with the two files for varying size context. It indicates that for English text choosing $N = 8192$ affords near-optimal compression without having the window unnecessarily large, while $N = 512$ is quite suitable for the picture file. Significantly better compression is achieved for the picture file.

Some techniques are sensitive to the value of M , and figure 6 shows the compression achieved for varying match size. The average match length for English text is about 3.5 characters and the picture file is 14 symbols. Choosing $M = 128$ results in a loss of only a few hundredths of bits per character loss for the picture and no effect on the compression of the file “book”, since matches of 128 characters do not occur. Choosing an unbounded M to obtain optimal compression results in very poor speed performance for the trie, binary tree, and Boyer-Moore.

Figure 7 plots the amount of memory required for each data structure (not including the Ziv-Lempel window) assuming that pointers and integers can be stored in two bytes. The most economical data structure is the linear search, which requires no extra memory. Least economical is list2, using a quarter of a megabyte, principally to store the two-dimensional arrays.

Figure 8 shows the average time taken to code each character for “book”, for each data structure that has been described, except the linear method variations. In general the data structures that require more storage achieve faster compression. Even the simple list1 method was up to 10 times faster than the linear search, although it is overtaken by the more sophisticated methods as N increases. The list2 method is particularly effective, even for large windows, because pairs of characters repeat infrequently, resulting in short lists to be searched. The hash method was about half as fast as list2, and suggests that the overhead of maintaining and searching several lists did not pay off. The hash method, however, uses considerably less memory. The binary search tree is more effective considering the amount of memory that it requires. The tree must have been relatively well balanced because the extra time taken by the splay tree to ensure balancing did not result in faster access time. The trie required about the same memory as the hash method, yet gave much worse performance; this can be attributed to the large proportion of time spent inserting each individual phrase into the trie.

The linear method variations are shown separately in Figure 9. Although Boyer-Moore sustains a high overhead in pre-processing the pattern in the lookahead buffer, the algorithm soon wins out as the window grows.

Figure 10 shows how the data structures performed for “picture.” The relative performances are significantly different from the English text because the picture file causes near worst-case behaviour in most of the data structures. Note that the scale in this graph is much larger than the one in Figure 8. The large runs of 0’s in the picture file will cause many collisions in the hash table and long chains of nodes in both the binary tree and the trie, and this is reflected in very slow compression. The self balancing splay tree has paid off in this situation, maintaining a performance advantage over the simple binary tree. List1 and list2 perform moderately well, although the negligible improvement of list2 over list1 indicates that the range of character pairs is not as rich as it is for English text. Boyer-Moore performed far better than all other methods for the picture because large repetitive sections are quickly scanned and no indexing data structure is maintained.²

The picture file is particularly testing, and it should be noted that such a file can be compressed quite successfully with $N = 512$. In this range most of the data structures give satisfactory performance, although they are still slower than when coding English text. There are various ways to avoid this slow encoding if such files are likely to be encoded frequently. Placing a smaller limit on the length of a match decreases the time spent comparing substrings, at the expense of losing a little compression. For the tree data structures, inserting only matched phrases – similar to the LZ78 variant LZFG [19] – instead of inserting every substring in the source, would reduce execution time with minimal compression loss.

Alternatively, with the binary search tree, the encoding speed could be monitored, and if it became unsatisfactory then splaying could be introduced to prevent poor performance. Another way of achieving this compromise between the two methods is to only splay nodes beyond some pre-specified depth in the tree. This achieves a compression speed between that of a simple binary tree and a full splay tree, avoiding the worst performance of either, but not doing as well as the best.

Conclusions

A surprisingly large selection of data structures is available to accelerate searching for longest matches. Each represents a trade-off between aver-

²A variation to the linked list method that searches for less likely symbols according to an adaptive model of symbol probabilities performs even better than Boyer-Moore for the picture file, but is unimpressive for most texts. [13]

age compression time, worst-case behaviour, and memory requirements. If storage is plentiful then the list2 method gives good performance, particularly for English text. A binary search tree gives good performance for only modest memory requirements, and splaying can be applied to avoid worst case performance. The trie, which would traditionally be the method of choice for this situation, does not appear to offer a particularly useful compromise between speed and storage, since it is outperformed on both counts by a binary search tree. The list1 method offers particularly good performance considering it requires very little storage. A hash table appears to be a contender, particularly with English text, and there is scope to improve its performance with a larger hash table, and by experimenting with the hash function and search strategy.

Worst case performance for most of the data structures is caused by files that contain long repeated substrings, such as an image that contains long runs of zero bits. Poor performance can be avoided by limiting the length of a match, and/or using a data structure, such as a splay tree, that does not depend on substrings of the text being random.

Acknowledgements

The authors are grateful to Bruce McKenzie, Alistair Moffat, Bob Kruse, and an anonymous referee for helpful comments on earlier versions of the paper.

References

- [1] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans Communications*, 32(4):395–402, 1984.
- [2] G. V. Cormack and R. N. Horspool. Data compression using dynamic markov modelling. *Computer Journal*, 30(6):541–550, 1987.
- [3] I. H. Witten, Neal R., and J. G. Cleary. Arithmetic coding for data compression. *Comm. ACM*, 30(6):520–540, 1987. Reprinted in *C Gazette* 2 (3) 4-25, (1987).
- [4] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

- [5] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans Information Theory*, 23(3):337–343, 1977.
- [6] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans Information Theory*, 24(5):530–536, 1978.
- [7] J.A. Storer and T. G. Szymanski. Data compression via textual substitution. *J ACM*, 29(4):928–951, 1982.
- [8] T.C. Bell. Better OPM/L text compression. *IEEE Trans Communications*, 34(12):1176–1182, December 1986.
- [9] R. P. Brent. A linear algorithm for data compression. *Australian Computer Journal*, 19(2):64–68, 1987.
- [10] T.C. Bell. *A unifying theory and improvements for existing approaches to text compression*. PhD thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 1987.
- [11] D. E. Knuth, J.H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J Computing*, 6(2):323–350, 1977.
- [12] R. S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm ACM*, 20(10):762–772, 1977.
- [13] D. C. Kulp. Very fast pattern matching for highly repetitive text. Technical report, Department of Computer Science, University of Canterbury, 1992.
- [14] D. Morrison. PATRICIA – a practical algorithm to retrieve information coded in alphanumeric. *J ACM*, 15(4):514–534, 1968.
- [15] E.M. McCreight. A space-economical suffix tree construction algorithm. *J ACM*, 23(2):262–272, 1976.
- [16] M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J ACM*, 28(1):16–24, 1981.
- [17] T. Raita and Teuhola J. Predictive text compression by hashing. In *ACM Conference on Information Retrieval*, New Orleans, 1987.
- [18] D.S. Sleator and Tarjan R. E. Self-adjusting binary search trees. *J ACM*, 32(3):652–686, 1985.

- [19] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Comm ACM*, 32(4):490–505.
- [20] T. C. Bell and D. C. Kulp. Longest match string searching for Ziv-Lempel compression. *Software - Practice and Experience*. (to appear).
- [21] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.
- [22] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1988.

Name	Method	Memory Requirement (pointers and integers)
linear	linear search	0
Boyer-Moore	Boyer-Moore scanning algorithm	$q + M$
list1	linked list based on first character	$2q + N$
list2	linked list based on first two characters	$2q^2 + N$
trie	trie with coalesced chains	$8N$
hash	hash table storing strings of length 1, 2, 4, 8	$15N - 1 + q$
bintree	binary search tree	$3N$
splay	binary search tree with splaying	$3N$

Table 1: Data structures evaluated in experiments

List of Figures

1	The LZB sliding window	17
2	A trie index for the window in Figure 1	18
3	Coalescing the chains of the trie in Figure 2	19
4	A binary search tree for the window in Figure 1	20
5	Compression of LZB against N ($M=N$)	21
6	Compression of LZB against M	22
7	Memory used by the data structures against N	23
8	Time take to compress the file “book”, $M = 128$	24
9	Time taken to compress the file “book” using Linear and Boyer-Moore, $M = 128$	25
10	Time taken to compress the file “picture” using all meth- ods, $M = 128$	26

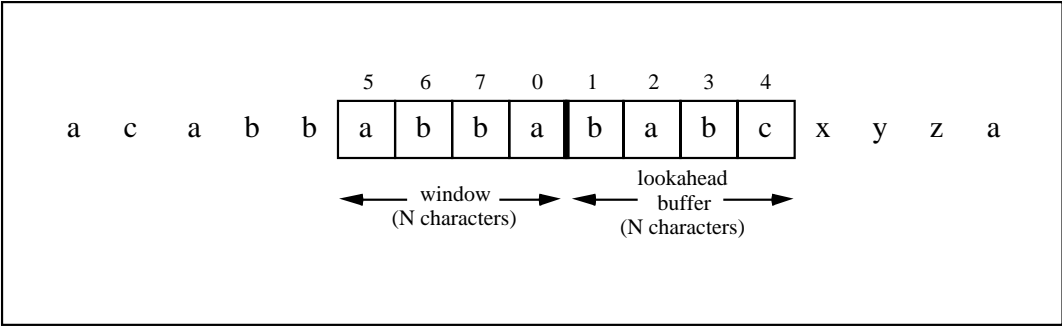


Figure 1: The LZB sliding window

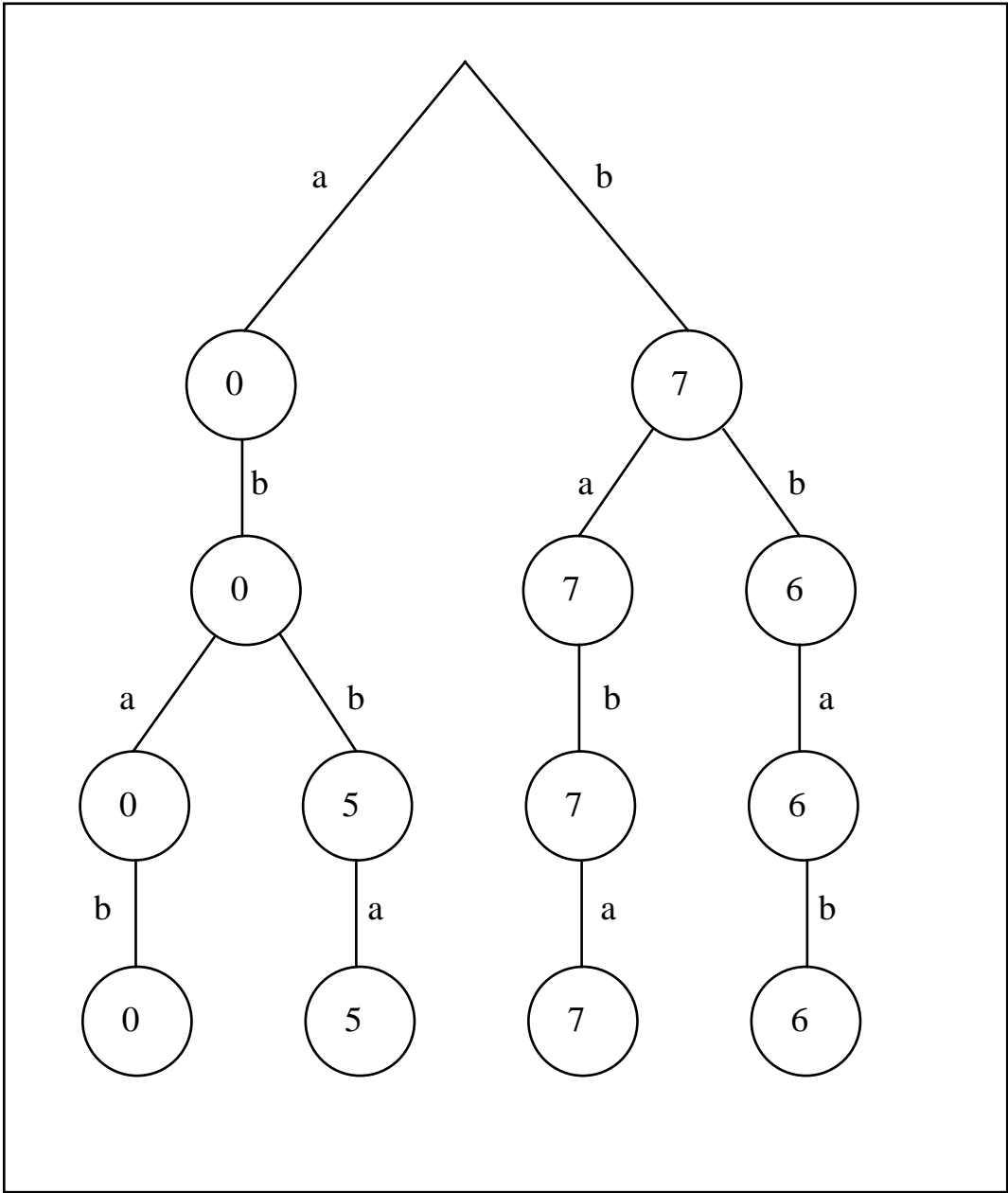


Figure 2: A trie index for the window in Figure 1

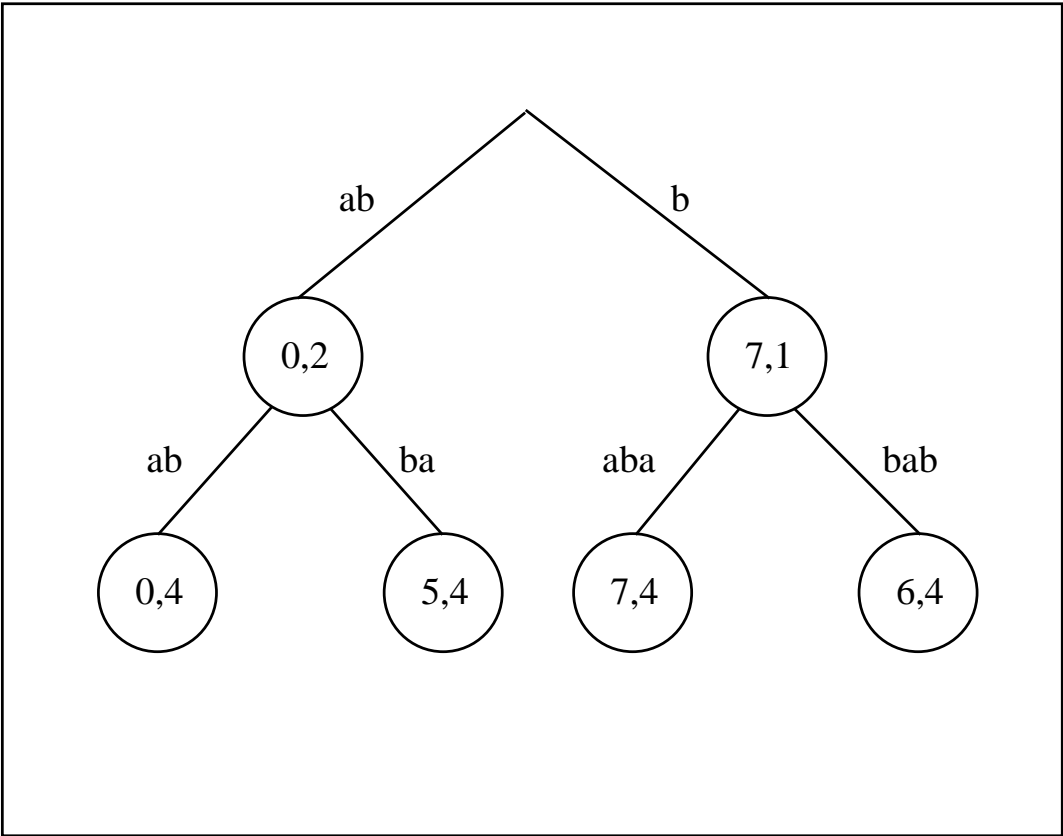


Figure 3: Coalescing the chains of the trie in Figure 2

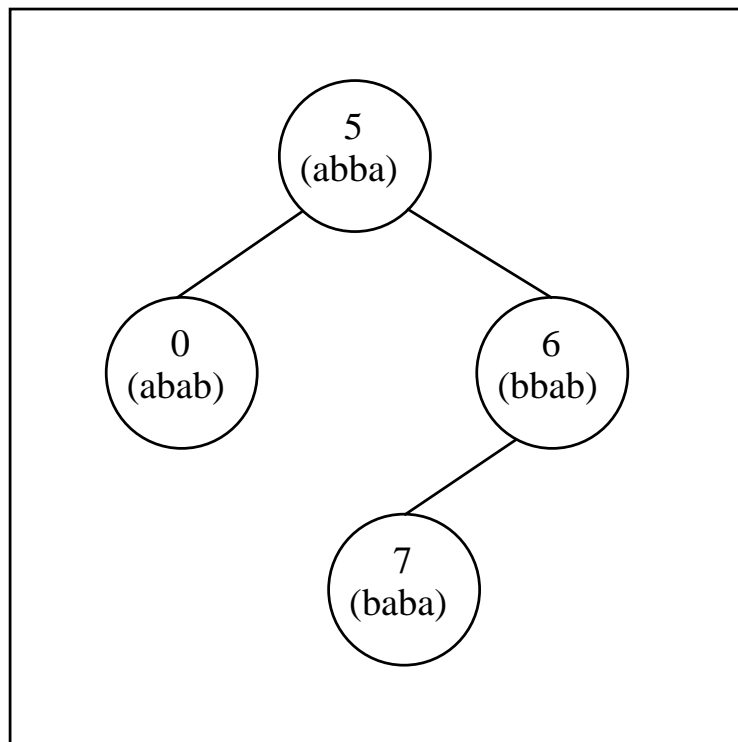


Figure 4: A binary search tree for the window in Figure 1

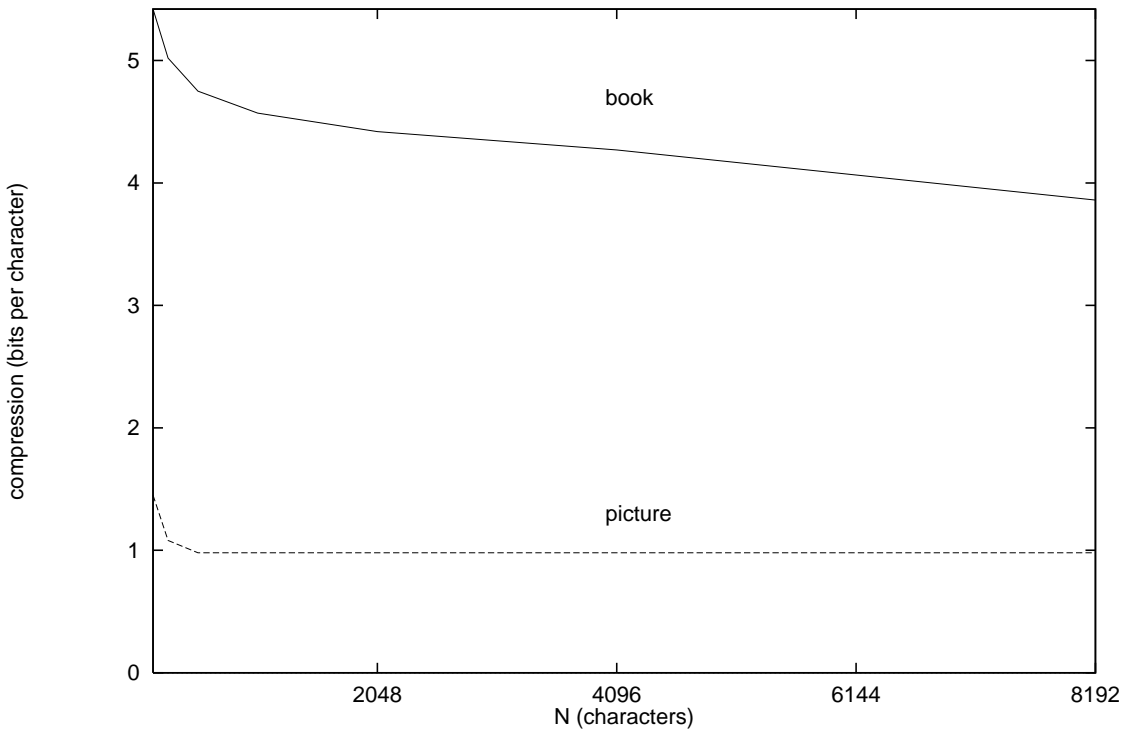


Figure 5: Compression of LZB against N ($M=N$)

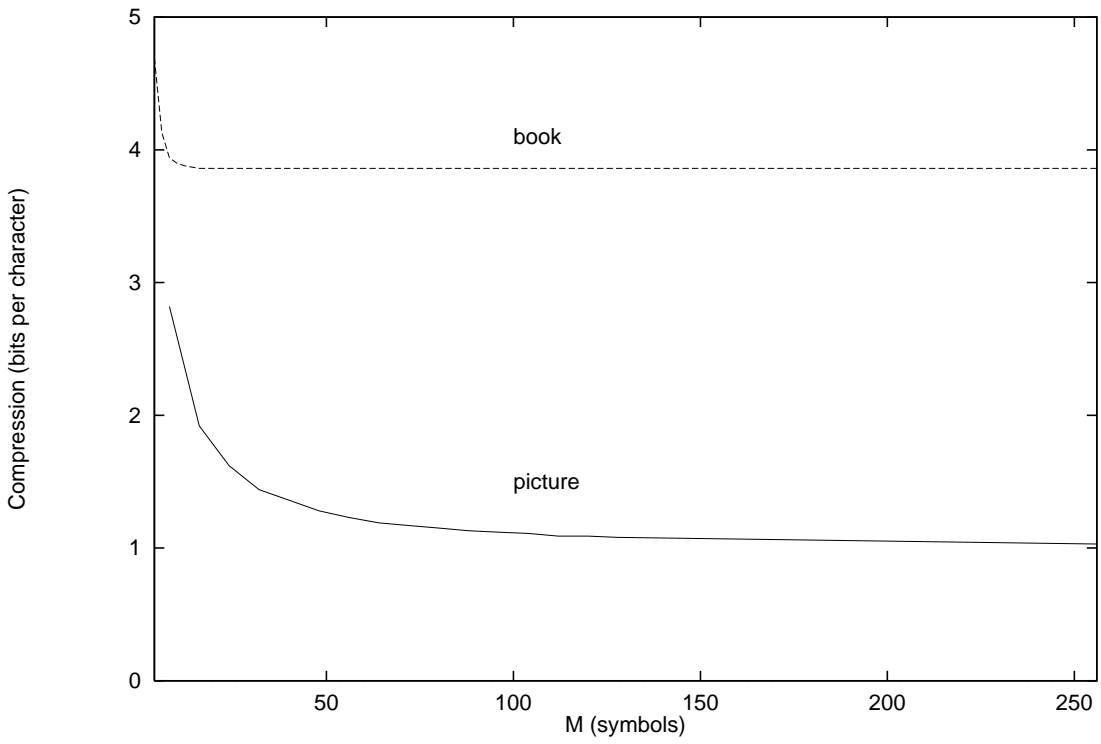


Figure 6: Compression of LZB against M

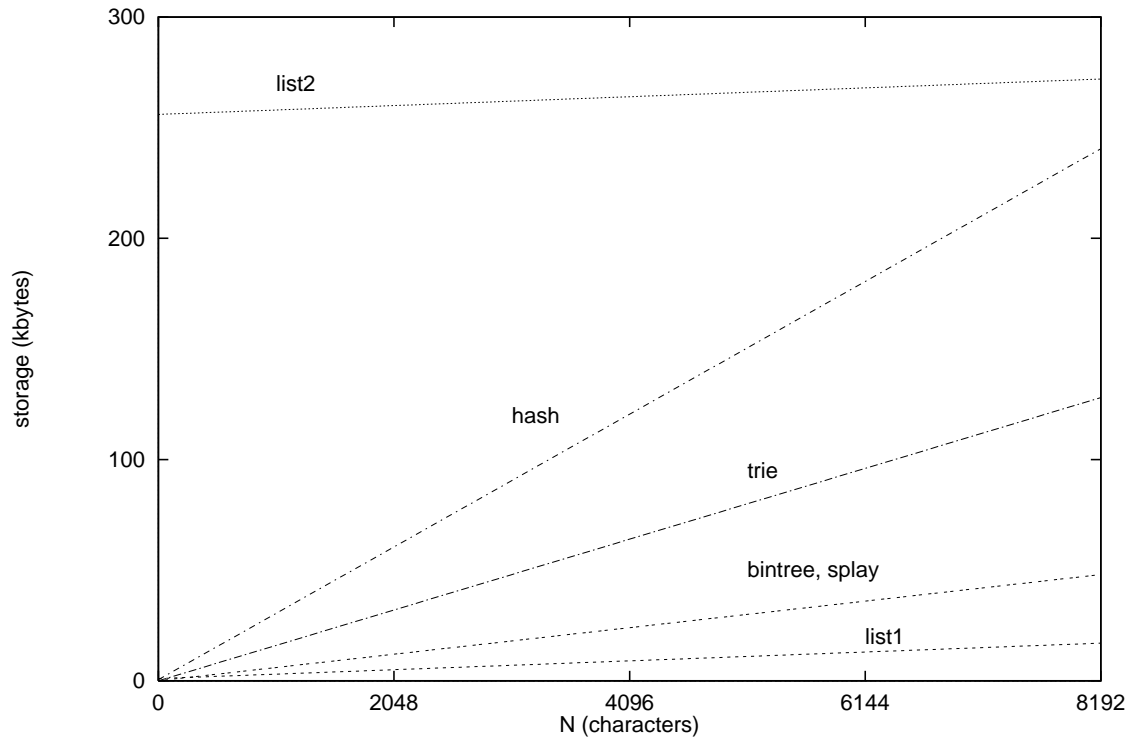


Figure 7: Memory used by the data structures against N

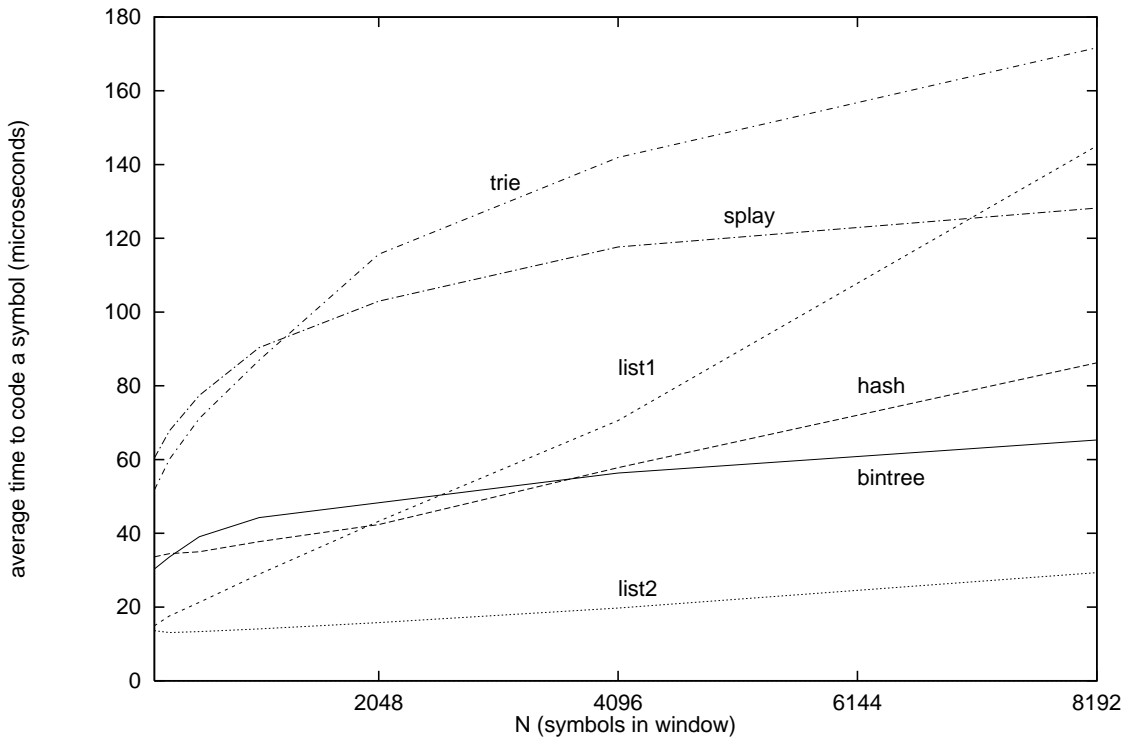


Figure 8: Time take to compress the file "book", $M = 128$

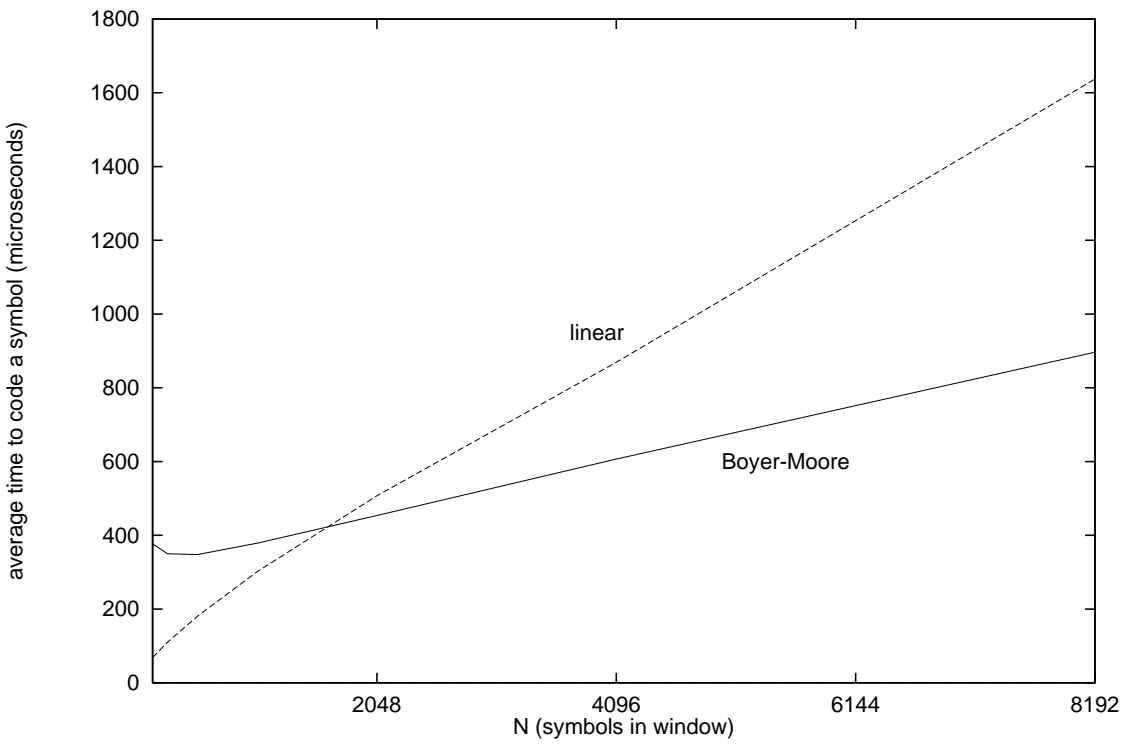


Figure 9: Time taken to compress the file “book” using Linear and Boyer-Moore, $M = 128$

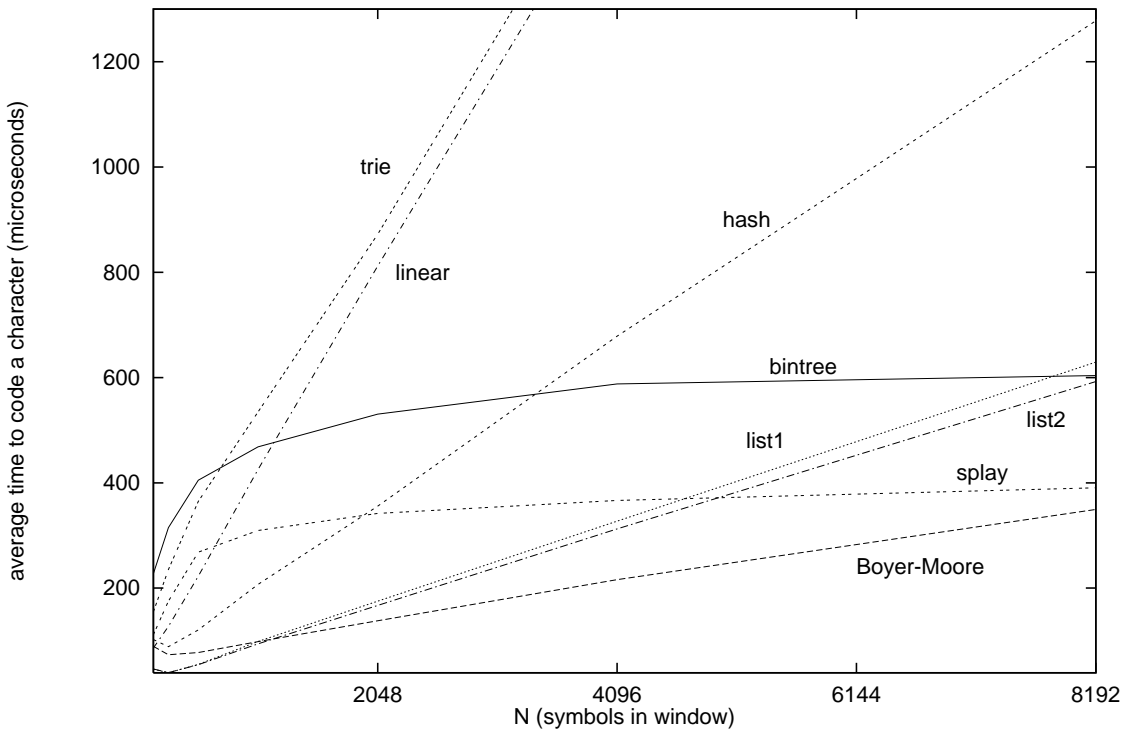


Figure 10: Time taken to compress the file “picture” using all methods, $M = 128$