The LZ-index: A Text Index Based on the Ziv-Lempel Trie

Gonzalo Navarro

Dept. of Computer Science, Univ. of Chile.
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.
Partially supported by Fondecyt Grant 1-020831.

Abstract

Let a text of u characters over an alphabet of size σ be compressible to n symbols by the LZ78 or LZW algorithm. We show how to build a data structure, called the LZ-index, based on the Ziv-Lempel trie that takes $4n \log_2 n(1 + o(1))$ bits of space (that is, 4 times the entropy of the text) and reports the R occurrences of a pattern of length m in worst case time $O(m^3 \log \sigma + (m+R) \log n)$. We present a practical implementation of the LZ-index with average search time $O(\sigma m \log u + \sqrt{uR})$. It turns out to be competitive with current alternatives when we take into consideration the time to report the positions or text contexts of the occurrences found. The implementation of the data structure is by itself an interesting lesson of theory versus practice.

1 Introduction and Related Work

A *text database* is a system providing fast access to a large mass of textual data. By far the most challenging requirement is that of performing fast text searching for user-entered patterns. The simplest (yet realistic and rather common) scenario is as follows. The text $T_{1...u}$ is regarded as a unique sequence of characters over an alphabet Σ of size σ, and the search pattern $P_{1...m}$ as another (short) sequence over Σ. Then the text search problem consists of finding all the R occurrences of P in T.

Modern text databases have to face two opposed goals. On the one hand, they have to provide fast access to the text. On the other, they have to use as little space as possible. The goals are opposed because, in order to provide fast access, an *index* has to be built on the text. An index is a data structure built on the text and stored in the database, hence increasing the space requirement. In recent years there has been much research on *compressed text databases*, focusing on techniques to represent the text and the index in succinct form, yet permitting efficient text searching.

Despite that there has been some work on succinct inverted indexes for natural language for a while [28, 24] (able of finding whole words and phrases), until a short time ago it was believed that any general index for string matching would need $\Omega(u)$ space. In practice, the smallest indexes available were the suffix arrays [20], requiring $u \log_2 u$ bits to index a text of u characters, which required $u \log_2 \sigma$ bits to be represented, so the index is in practice larger than the text (typically 4 times the text size).

Since the last decade, several attempts to reduce the space of the suffix trees [3] or arrays have been made by Kärkkäinen and Ukkonen [12, 15], Kurtz [17], Mäkinen [19], and Abouelhoda, Ohlebusch and Kurtz [1], obtaining remarkable improvements, albeit no spectacular ones. Moreover, they have concentrated on the space requirement of the data structure only, needing the text separately available.

The first achievement of a new trend started with Grossi and Vitter [9], who presented a suffix array compression method for binary texts, which needed $O(u)$ bits and was able to report all the R occurrences of P in T in $O\left(\frac{m}{\log u} + (R+1)\log^{\varepsilon} u\right)$ time. However, they need the text as well as the index in order to answer queries.

Following this line, Sadakane [25] presented a suffix array compression method for general texts (not only binary) that requires $u\left(\frac{1}{\varepsilon}H_0 + 8 + 3\log_2 H_0\right)(1+o(1)) + \sigma\log_2\sigma$ bits, where H_0 is the zero-order entropy of the text. This index can search in time $O(m\log u + R\log^{\varepsilon} u)$ and contains enough information to reproduce the text: any piece of text of length L is obtained in $O(L + \log^{\varepsilon} u)$ time. This means that the index *replaces* the text, which can hence be deleted. This is an *opportunistic* scheme, i.e., the index takes less space if the text is compressible. Yet there is a minimum of $8u$ bits of space which has to be paid independently of the entropy of the text.

Ferragina and Manzini [6] presented a different approach to compress the suffix array based on the Burrows-Wheeler transform and block sorting. They need $5uH_k + O\left(u\frac{\log\log u + \sigma\log\sigma}{\log u}\right)$ bits and can answer queries in $O(m + R\log^{\varepsilon} u)$ time, where H_k is the k-th order entropy and the formula is valid for any constant k. This scheme is also opportunistic. However, there is a large constant $\sigma\log\sigma$ involved in the sublinear part which does not decrease with the entropy, and a huge additive constant larger than σ^{σ}. (In a real implementation [7] they removed these constants at the price of a not guaranteed search time.)

Recently, Sadakane [26] has proposed a compact suffix array representation that includes longest common prefix information, which is able to count the occurrences of P in $O(m)$ time and of traversing the suffix tree in $O(n\log^{\varepsilon} n)$ time. It needs $\frac{1}{\varepsilon}nH_1 + O(n)$ bits. Its main interest lies in its ability to handle large alphabets, where it is superior to [6].

However, there are older attempts to produce succinct indexes, by Kärkkäinen and Ukkonen [14, 13]. Their main idea is to use a suffix tree that indexes only the beginnings of the blocks produced by a Ziv-Lempel compression (see next section if not familiar with Ziv-Lempel). This is the only index we are aware of which is based on this type of compression. In [13] they obtain a range of space-time trade-offs. The smallest indexes need $O\left(u\left(\log\sigma + \frac{1}{\varepsilon}\right)\right)$ bits, i.e., the same space of the original text, and are able to answer queries in $O\left(\frac{\log\sigma}{\log u}m^2 + m\log u + \frac{1}{\varepsilon}R\log^{\varepsilon} u\right)$ time. Note, however, that this index is not opportunistic, as it takes space proportional to the text, and indeed needs the text besides the data of the index.

In this paper we propose a new index on these lines, called the LZ-index. Instead of using a generic Ziv-Lempel algorithm, we stick to the LZ78/LZW format and its specific properties. We do not build a suffix tree on the strings produced by the LZ78 algorithm. Rather, we use the very same LZ78 trie that is produced during compression, plus other related structures. We borrow some ideas from Kärkkäinen and Ukkonen's work, but in our case we have to face additional complications because the LZ78 trie has less information than the suffix tree of the blocks. As a result, our index is smaller but has a higher search time. If we call n the number of blocks in the compressed text, then our index takes $4n \log_2 n(1+o(1))$ bits of space and answers queries in $O(m^3 \log \sigma + (m+R) \log n)$ time. It is shown in [16, 8] that Ziv-Lempel compression asymptotically approaches H_k for any k. Since this compressed text needs at least $n \log_2 n$ bits of storage, we have that our index is opportunistic, taking at most $4uH_k$ bits, for any k.

This representation, moreover, contains the information to reproduce the text. We can reproduce a text context of length L around an occurrence found (and in fact any sequence of blocks) in $O(L \log \sigma)$ time, or obtain the whole text in time $O(u \log \sigma)$. The index can be built in $O(u \log \sigma)$ time. Finally, the time can be reduced to $O(m^3 \log \sigma + m \log n + R \log^{\varepsilon} n)$ provided we pay $O\left(\frac{1}{\varepsilon} n \log n\right)$ space.

About at the same time and independently of us [8], Ferragina and Manzini have proposed another idea combining compressed suffix arrays and Ziv-Lempel compression. They achieve optimal $O(m+R)$ search time at the price of $O(uH_k \log^{\varepsilon} u)$ space. Moreover, this space includes two compressed suffix arrays of the previous type [6] and their large constant terms. It is interesting that they share, like us, several ideas of previous work on sparse suffix trees [14, 13].

What is unique in our approach is the reconstruction of the occurrences using a data structure that does not record full suffix information but just of text substrings, thus addressing the problem of reconstructing pattern occurrences from these pieces information.

In addition to our theoretical proposal, we have implemented our index. The implementation involves several considerations on the practicality of the theoretical decisions, which offer worst-case big-O guarantees but do not care for the constants, which are important in practice. The final prototype was tested on large natural language and DNA texts. It takes about 5 times the space needed by the compressed text (which is close to our prediction $4uH_k$). On a 2 GHz Pentium IV machine, the index is built at a rate of 1–2 Mb/sec (which is competitive with current technology) and uses a temporary extra space similar to a suffix array construction (5 times the text size, which is large but usual, and can be reduced in 50% by standard means). On a 50 Mb text, a normal query takes 2 to 4 milliseconds (msecs), depending linearly on its length, plus the time to report the R occurrences, at a rate of 600–800 per msec. Text lines can be displayed at a rate of 14 lines per msec.

We have compared our index against existing alternatives. Although our index is much slower to *count* how many occurrences are there, it is much faster to *report* their position or their text context. Indeed, we show that if there are more than 300–1,400 occurrence positions to report (this depends on the text type), then our index is faster than the others. This number goes down to 13–65 if the text lines of the occurrences have to be shown. Being able of reproducing the text is an essential feature, since all these indexes *replace* the text and hence our only way to see the text is asking them to reproduce it.

This paper is organized as follows. In Section 2 we explain the Ziv-Lempel compression. In Section 3 we present the basic ideas of our technique. Section 4 explains how to represent the data structures we use in succint space. Section 5 gives a theoretical analysis of the data structure, in terms of space, construction and query time. Section 6 describes the practical implementation of the index. Section 7 compares our implementation against the most prominent alternatives. Section 8 gives our conclusions and future work directions.

2 Ziv-Lempel Compression

The general idea of Ziv-Lempel compression is to replace substrings in the text by a pointer to a previous occurrence of them. If the pointer takes less space than the string it is replacing, compression is obtained. Different variants over this type of compression exist, see for example [4]. We are particularly interested in the LZ78/LZW format, which we describe in depth.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [29]) is based on a dictionary of blocks, in which we add every new block computed. At the beginning of the compression, the dictionary contains a single block b_0 of length 0. The current step of the compression is as follows: if we assume that a prefix $T_{1\ldots j}$ of T has been already compressed in a sequence of blocks $Z = b_1 \ldots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1\ldots u}$ which is a block of the dictionary. Once we have found this block, say b_s of length ℓ_s, we construct a new block $b_{r+1} = (s, T_{j+\ell_s+1})$, we write the pair at the end of the compressed file Z, i.e $Z = b_1 \ldots b_r b_{r+1}$, and we add the block to the dictionary. It is easy to see that this dictionary is prefix-closed (i.e. any prefix of an element is also an element of the dictionary) and a natural way to represent it is a trie.

We show in Figure 1 the compression of the text *alabar a la alabarda para apalabrarla*[1], which will be our running example. For readability we have changed the space to underscore and have assumed its code is larger than those of normal letters.

The first block is $(0, a)$, and next $(0, l)$. When we read the next a, a is already block 1 in the dictionary, but ab is not in the dictionary. So we create a third block $(1, b)$. We then read the next a, a is already block 1 in the dictionary, but ar does not appear. So we create a new block $(1, r)$, and so on. The full compressed text is

$$(0,a)\ (0,l)\ (1,b)\ (1,r)\ (0,\_)\ (1,\_)\ (2,a)\ (5,a)\ (7,b)\ (4,d)\ (6,p)\ (4,a)\ (8,p)\ (1,l)\ (3,r)\ (4,l)\ (1,\$)$$

were we have added a terminator character "\$", smaller than any other character, to ensure that every block corresponds to a different node.

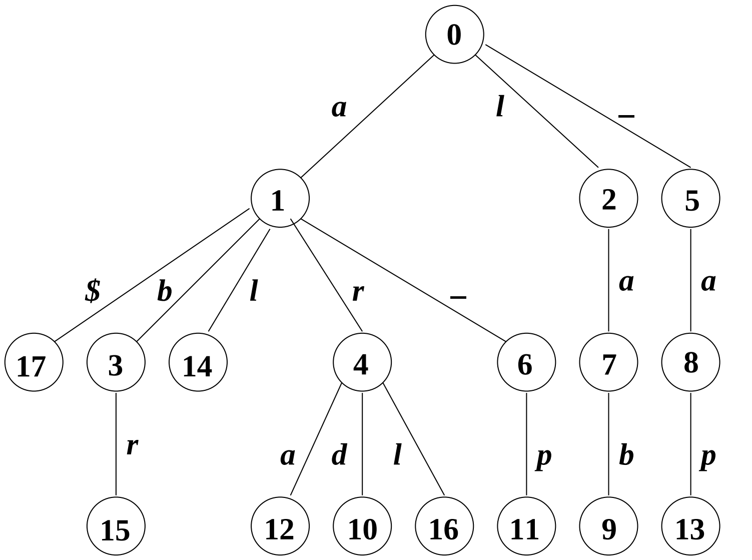
The compression algorithm is $O(u)$ time in the worst case and efficient in practice if the dictionary is stored as a trie, which allows rapid searching of the new text prefix (for each character of T we move once in the trie). The decompression needs to build the same dictionary (the pair that defines the block r is read at the r-th step of the algorithm).

Many variations on LZ78 exist, which deal basically with the best way to code the pairs in the compressed file. A particularly interesting variant is from Welch, called LZW [27]. In this case, the extra letter (second element of the pair) is not coded, but it is taken as the first letter of the next block (the dictionary is started with one block per letter). LZW is used by Unix's *Compress* program.

In this paper we do not consider LZW separately but just as a coding variant of LZ78. This is because the final letter of LZ78 can be readily obtained by keeping count of the first letter of each block (this is copied directly from the referenced block) and then looking at the first letter of the next block.

An interesting property of this compression format is that every block represents a different text substring. The only possible exception is the last block. We use this property in our algorithm,

[1] A not totally meaningful Spanish phrase, but one with nice periodicity properties!



alabar a la alabarda para apalabrarla

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	*l*	*ab*	*ar*	_	*a_*	*la*	*_a*	*lab*	*ard*	*a_p*	*ara*	*_ap*	*al*	*abr*	*arl*	*a$*

Figure 1: Ziv-Lempel trie and parse for our running example. For example, block number 10 represents string *ard*, which is spelled out when we move from the trie root to node labeled 10.

and deal with the exception by adding a special character "\$" (not in the alphabet) at the end of the text. The last block will contain this character and thus will be unique too.

Another concept that is worth reminding is that a set of strings can be lexicographically sorted, and we call the *rank* of a string its position in the lexicographically sorted set. Moreover, if the set is arranged in a trie data structure, then all the strings represented in a subtree form a lexicographical interval of the universe. We remind that, in lexicographic order, $\varepsilon \leq x$, $ax \leq by$ if $a < b$, and $ax \leq ay$ if $x \leq y$, for any strings x, y and characters a, b.

3 Basic Technique

We now present the basic idea to search for a pattern $P_{1\ldots m}$ in a text $T_{1\ldots u}$ which has been compressed using the LZ78 or LZW algorithm into $n+1$ blocks $T = B_0 \ldots B_n$, such that $B_0 = \varepsilon$; $\forall k \neq \ell,\ B_k \neq B_\ell$ (that is, no two blocks are equal); and $\forall k \geq 1,\ \exists \ell < k, c \in \Sigma,\ B_k = B_\ell \cdot c$ (that is, every block except B_0 is formed by a previous block plus a letter at the end).

3.1 Data Structures

We start by defining the data structures used, without caring for the exact way they are represented. The problem of their succinct representation, and consequently the space occupancy and time complexity, is considered in Section 4.

1. $LZTrie$: is the trie formed by all the blocks $B_0 \ldots B_n$. Given the properties of LZ78 compression, this trie has exactly $n+1$ nodes, each one corresponding to a string. $LZTrie$ stores enough information so as to permit the following operations on every node x:

   (a) $id_t(x)$ gives the node identifier, i.e., the number k such that x represents B_k;

   (b) $leftrank_t(x)$ and $rightrank_t(x)$ give the minimum and maximum lexicographical position of the blocks represented by the nodes in the subtree rooted at x, among the set $B_0 \ldots B_n$;

   (c) $parent_t(x)$ gives the tree position of the parent node of x; and

   (d) $child_t(x, c)$ gives the tree position of the child of node x by character c, or $null$ if no such child exists.

   Additionally, the trie must implement the operation $rth_t(rank)$, which given a rank r gives the node representing the r-th string in $B_0 \ldots B_n$ in lexicographical order. Figure 2 shows the $LZTrie$ data structure for our running example.

2. $RevTrie$: is the trie formed by all the reverse strings $B_0^r \ldots B_n^r$. For this structure we do not have the nice properties that the LZ78/LZW algorithm gives to $LZTrie$: there could be internal nodes not representing any block. We need the same operations for $RevTrie$ than for $LZTrie$, which are called id_r, $leftrank_r$, $rightrank_r$, $parent_r$, $child_r$ and rth_r.

   Figure 3 shows the $RevTrie$ data structure for our running example.

3. $Node$: is a mapping from block identifiers to their node in $LZTrie$.

4. $Range$: is a data structure for two-dimensional searching in the space $[0 \ldots n] \times [0 \ldots n]$. The points stored in this structure are $\{(revrank(B_k^r), rank(B_{k+1})),\ k \in 0 \ldots n-1\}$, where $revrank$ is the lexicographical rank in $B_0^r \ldots B_n^r$ and $rank$ is the lexicographical rank in $B_0 \ldots B_n$. For each such point, the corresponding k value is stored.

   Figure 4 shows the $Range$ data structure for our running example.

3.2 Search Algorithm

Let us consider the search process now. We distinguish three types of occurrences of P in T, depending on the block layout (see Figure 5):

(a) the occurrence lies inside a single block;

(b) the occurrence spans two blocks, B_k and B_{k+1}, such that a prefix $P_{1 \ldots i}$ matches a suffix of B_k and the suffix $P_{i+1 \ldots m}$ matches a prefix of B_{k+1}; and

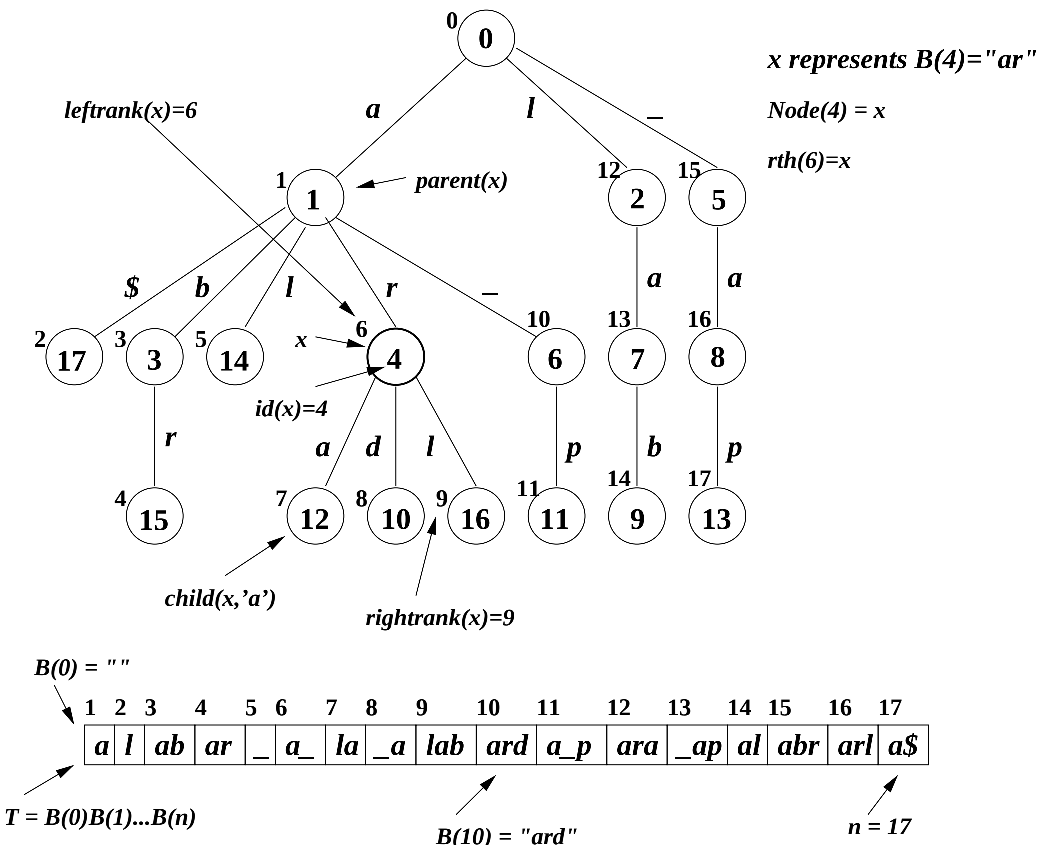


Figure 2: $LZTrie$ data structure for our running example. The numbers over the nodes are their rank. We show the values that correspond to node x, which represents block number 4 and is the 6th string in the set.

(c) the occurrence spans three or more blocks, $B_k \dots B_\ell$, such that $P_{i\dots j} = B_{k+1} \dots B_{\ell-1}$, $P_{1\dots i-1}$ matches a suffix of B_k and $P_{j+1\dots m}$ matches a prefix of B_ℓ.

Note that each possible occurrence of P lies exactly in one of the three cases above. We explain now how each type of occurrence is found.

3.2.1 Occurrences Lying Inside a Single Block

Given the properties of LZ78/LZW, every block B_k containing P is formed by a shorter block B_ℓ concatenated to a letter c. If P does not occur at the end of B_k, then B_ℓ contains P as well. We want to find the shortest possible block B in the referencing chain for B_k that contains the occurrence of P. This block B finishes with the string P, hence it can be easily found by searching for P^r in $RevTrie$.

Hence, in order to detect all the occurrences that lie inside a single block we do as follows:

1. Search for P^r in $RevTrie$. We arrive at a node x such that every string stored in the subtree

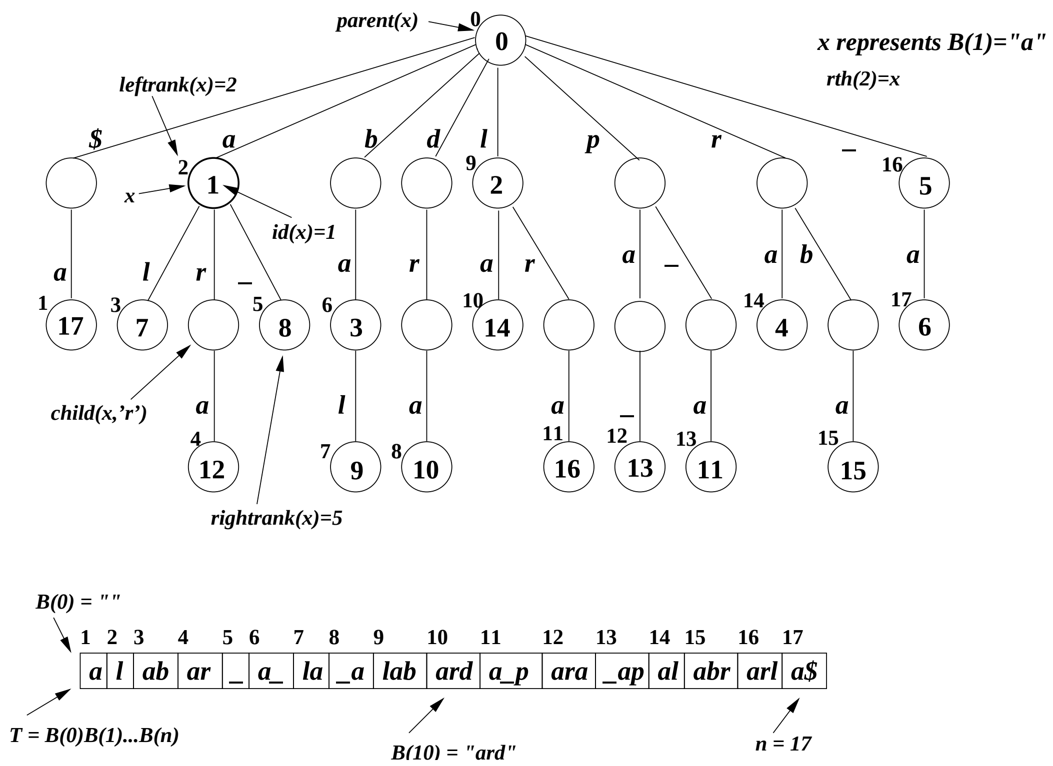


Figure 3: $RevTrie$ data structure for our running example. As we store the reversed strings, the set is not prefix-closed and not every node corresponds to a block identifier. We show the values that correspond to node x, which represents block number 1 and is the 2nd string in the set.

rooted at x represents a block ending with P.

2. Evaluate $leftrank_r(x)$ and $rightrank_r(x)$, obtaining the lexicographical interval (in the reversed blocks) of blocks finishing with P.

3. For every rank $r \in leftrank_r(x) \ldots rightrank_r(x)$, obtain the corresponding node in $LZTrie$, $y = Node(rth_r(r))$. Now we have identified the nodes in the normal trie that finish with P and have to report all their extensions, i.e., all their subtrees.

4. For every such y, traverse all the subtree rooted at y and report every node found. In this process we can know the exact distance between the end of P and the end of the block. Note that a single block containing several occurrences will report each of them, since we will report a subtree that is contained in another subtree reported.

Figure 6 illustrates the first part on our running example. Assume we search for ab. We look for ba on $RevTrie$ and reach the highlighted node. With $leftrank$ and $rightrank$ we find that the lexicographical range corresponding to its subtree is $[6 \ldots 7]$. For each such position we use rth_r to determine the block identifier, so as to obtain the list of identifiers of the subtree, $\{3, 9\}$.

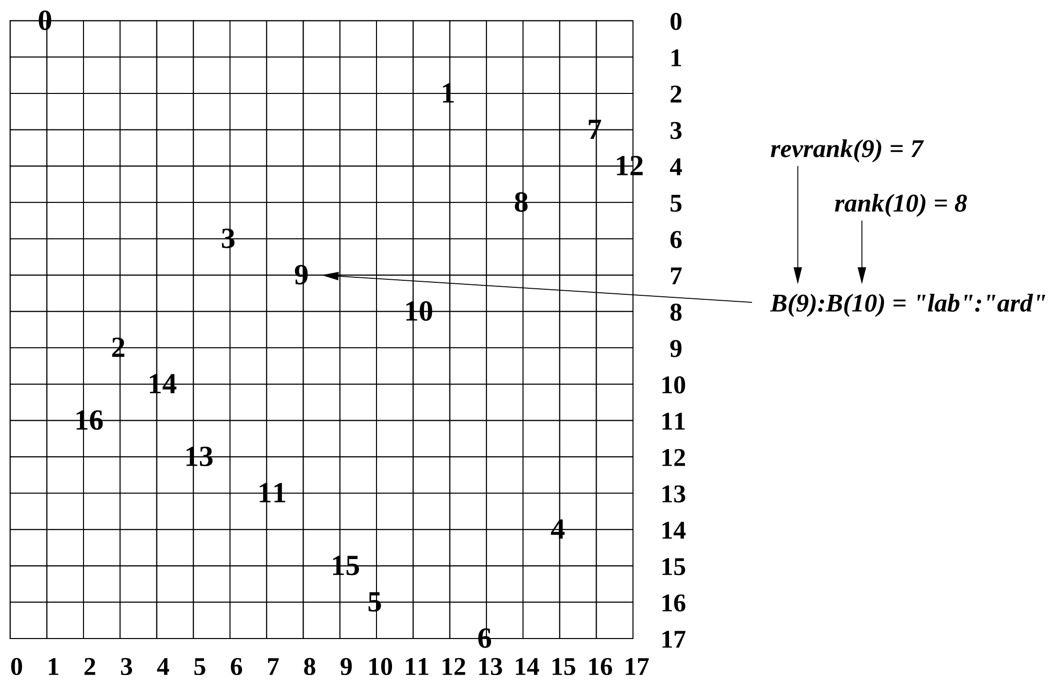


Figure 4: *Range* data structure for our running example. For instance, the pair of consecutive blocks 9:10 have reversed rank and rank, respectively, 7 and 8. Hence block number 9 is stored at row 7 and column 8 of the data structure.

Figure 7 shows the second part of the search on our running example. For each block in the list $\{3, 9\}$, we use $Node$ to find the corresponding node in $LZTrie$, and report all the subtrees. Hence block 3 leads us to report also block 15, while block 9 just reports itself. It is easy to deduce the offset in the reported blocks, counting from the end: the nodes in the list have offset m to the end of the block, their children $m+1$, their grandchildren $m+2$, and so on.

3.2.2 Occurrences Spanning Two Blocks

P can be split at any position, so we have to try them all. The idea is that, for every possible split, we search for the reverse pattern prefix in $RevTrie$ and the pattern suffix in $LZTrie$. Now we have two ranges, one in the space of reversed strings (i.e., blocks finishing with the first part of P) and

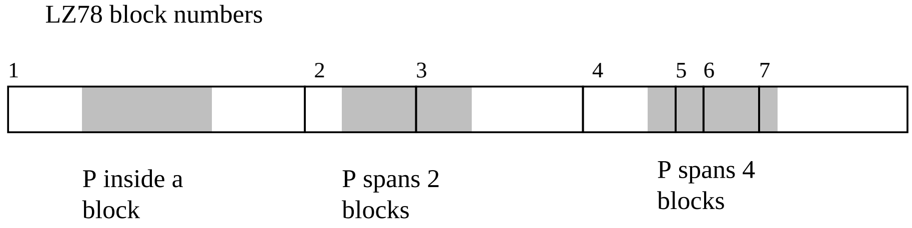


Figure 5: Different situations in which P can match inside T.

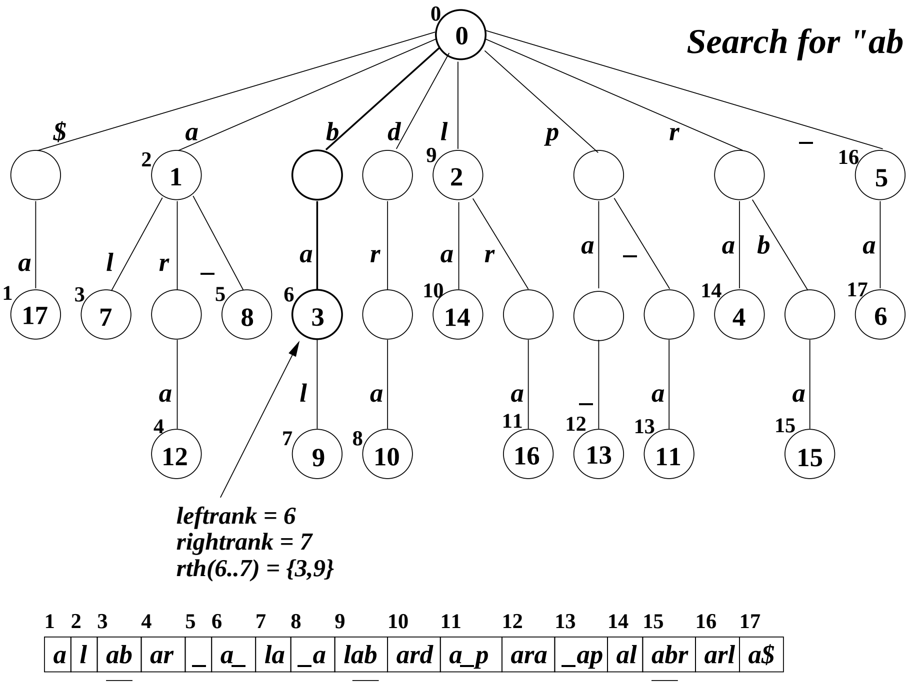


Figure 6: Reporting occurrences of type 1 of $P = ab$ in our running example, first part.

one in that of the normal strings (i.e. blocks starting with the second part of P), and need to find the pairs of blocks $(k, k+1)$ such that k is in the first range and $k+1$ is in the second range. This is what the range searching data structure is for. Hence the steps are:

1. For every $i \in 1 \ldots m-1$, split P in $pref = P_{1\ldots i}$ and $suff = P_{i+1\ldots m}$ and do the next steps.

2. Search for $pref^r$ in $RevTrie$, obtaining x. Search for $suff$ in $LZTrie$, obtaining y.

3. Search for the range $[leftrank_r(x) \ldots rightrank_r(x)] \times [leftrank_t(y) \ldots rightrank_t(y)]$ using the $Range$ data structure.

4. For every pair $(k, k+1)$ found, report k. We know that P_i is aligned at the end of B_k.

Figure 8 exemplifies the first part on our running example. Assume we search for ala (we will find only its occurrences of type 2). We look for the suffixes a and la on $LZTrie$, reaching the highlighted nodes. With $leftrank$ and $rightrank$ we find that their ranges are [1,9] and [13,14], respectively.

Figure 9 shows the second part. We search for the reverse prefixes of ala, namely la and a, in $RevTrie$. The nodes reached are highlighted. Their ranges are, respectively, [10,10] and [2,5].

Finally, Figure 10 shows the last part of the search. We join prefix a with suffix la, obtaining a 2-dimensional rank range (2,13):(5,14); and prefix al with suffix a, obtaining a 2-dimensional range (10,1):(10,9). Both ranges are searched for in $Range$, and all the block identifiers found are reported. The offsets are known from the splitting point.

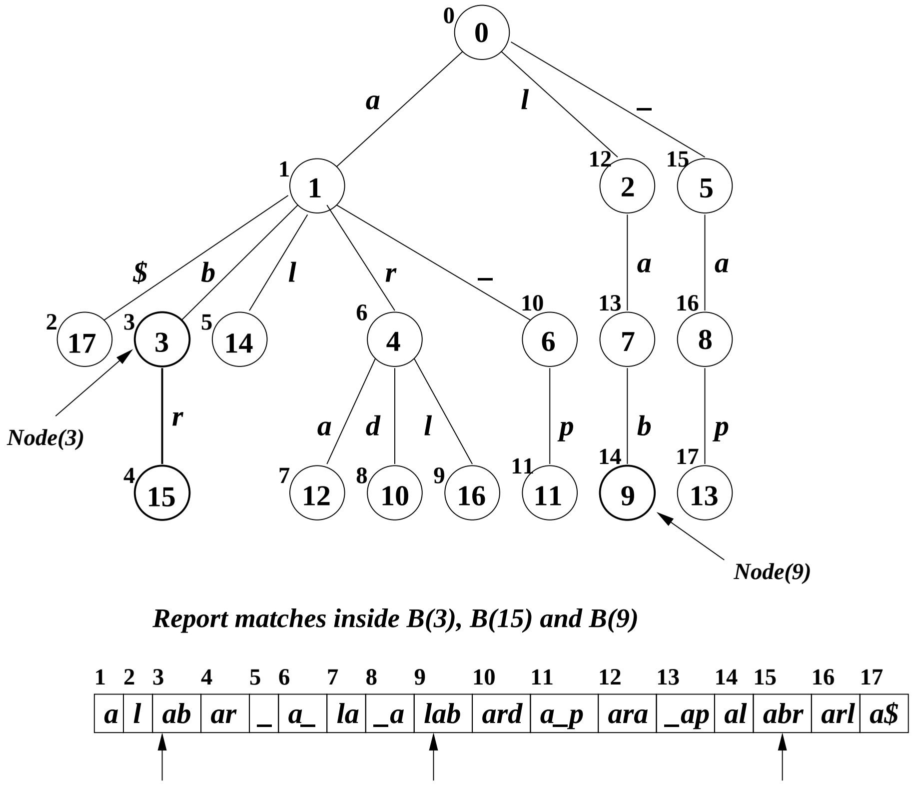


Figure 7: Reporting occurrences of type 1 of $P = ab$ in our running example, second part.

3.2.3 Occurrences Spanning Three Blocks or More

We need one more observation for this part. Recall that the LZ78/LZW algorithm guarantees that every block represents a different string. Hence, there is at most one block matching $P_{i...j}$ for each choice of i and j. This fact severely limits the number of occurrences of this class that may exist.

The idea is, first, to identify the only possible block that matches every substring $P_{i...j}$. We store the block numbers in m arrays A_i, where A_i stores the blocks corresponding to $P_{i...j}$ for all j. Then, we try to find concatenations of successive blocks B_k, B_{k+1}, etc. that match contiguous pattern substrings. Again, there is only one candidate (namely B_{k+1}) to follow an occurrence of B_k in the pattern. Finally, for each maximal concatenation of blocks $P_{i...j} = B_k \ldots B_\ell$ contained in the pattern, we determine whether B_{k-1} finishes with $P_{1...i-1}$ and $B_{\ell+1}$ starts with $P_{j+1...m}$. If this is the case we can report an occurrence. Note that there cannot be more than $O(m^2)$ occurrences of this type. So the algorithm is as follows:

1. For every $1 \le i \le j \le m$, search for $P_{i...j}$ in $LZTrie$ and record the node x found in $C_{i,j} = x$, as well as add $(id_t(x), j)$ to array A_i. The search is made for increasing i and for each i value we increase j. This way we perform a single search in the trie for each i. If there is no node corresponding to $P_{i...j}$ we stop searching and adding entries to A_i, and store null values in $C_{i,j'}$ for $j' \ge j$. At the end of every i-turn, we sort A_i by block number. Mark every $C_{i,j}$ as *unused*.

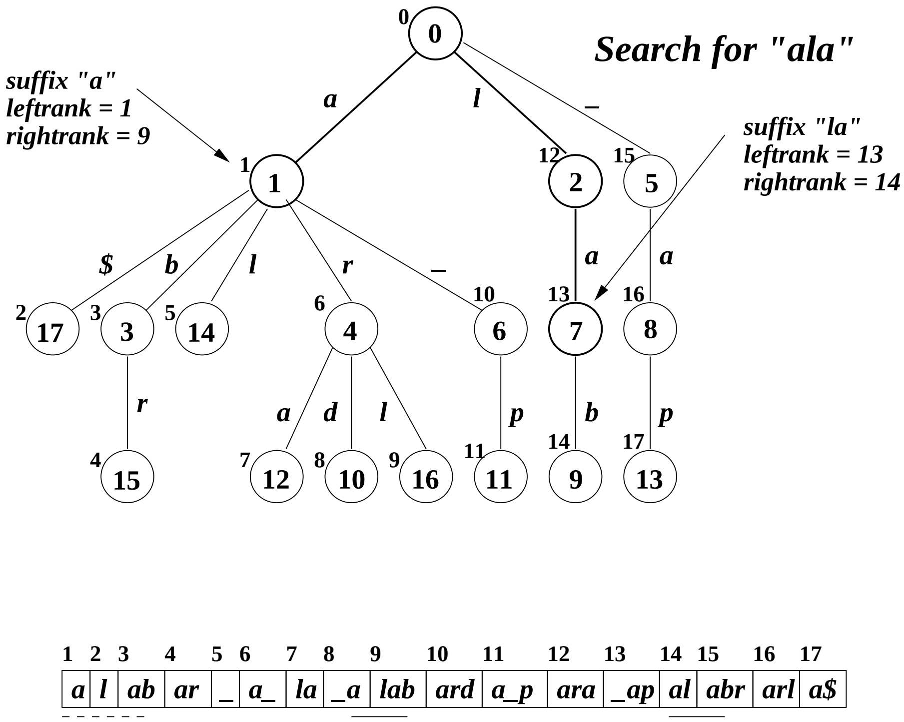


Figure 8: Reporting occurrences of type 2 of $P = ala$ in our running example, first part.

2. For every $1 \le i \le j < m$, for increasing j, try to extend the match of $P_{i...j}$ to the right. We do not extend to the left because this, if useful, has been done already (we mark used ranges to avoid working on a sequence that has been tried already from the left). Let S and S_0 denote $id_t(C_{i,j})$, and find $(S+1, r)$ in A_{j+1}. If r exists, mark $C_{j+1,r}$ as *used*, increment S and repeat the process from $j = r$. Stop when the occurrence cannot be extended further (no such r is found).

   (a) For each maximal occurrence $P_{i...r}$ found ending at block S such that $r < m$, check whether block $S+1$ starts with $P_{r+1...m}$, i.e., whether $leftrank_t(Node(S+1)) \in leftrank_t(C_{r+1,m}) \ldots rightrank_t(C_{r+1,m})$. Note that $leftrank_t(Node(S+1))$ is the exact rank of node $S+1$, since every internal node is the first among the ranks of its subtree. Note also that there cannot be an occurrence if $C_{r+1,m}$ is null. If $r < m$ and block $S+1$ does not start with $P_{r+1...m}$, then stop here and move to the next maximal occurrence.

   (b) If $i > 1$, then check whether block $S_0 - 1$ finishes with $P_{1...i-1}$. For this sake, find $Node(S_0 - 1)$ and use the $parent_t$ operation to check whether the last $i-1$ nodes, read backward, equal $P^r_{1...i-1}$. If $i > 1$ and block $S_0 - 1$ does not finish with $P_{1...i-1}$, then stop here and move to the next maximal occurrence.

   (c) Report node $S_0 - 1$ as the one containing the beginning of the match. We know that P_{i-1} is aligned at the end of this block.

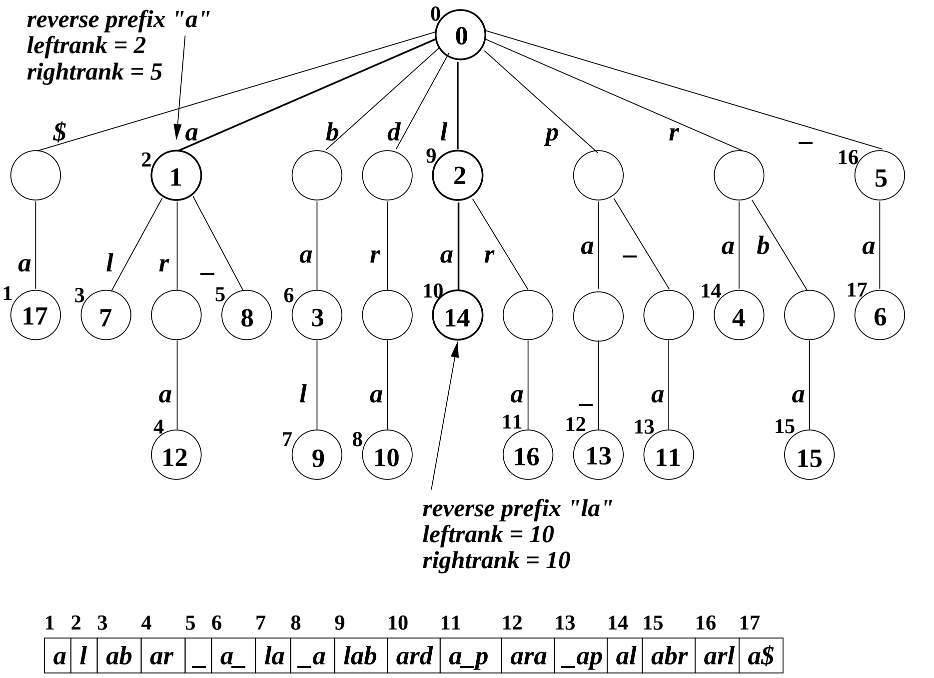


Figure 9: Reporting occurrences of type 2 of $P = ala$ in our running example, second part.

Note that we have to make sure that the occurrences reported span at least 3 blocks.

Figure 11 exemplifies the first part on our running example. Assume we search for *alaba*. We look for all the substrings of P and fill matrix C and the A vectors.

Figure 12 shows the second part. We obtain the maximal occurrences from the A vectors. In our example, we could join blocks B_1 to B_3 in a single maximal occurrence.

Figure 13 shows the third part of the search. We check that the maximal occurrences continue appropriately to the end of the pattern. Three maximal occurrences pass the test, for example $B_1 \ldots B_3 = P_{1\ldots4}$, since $Node(4)$ is below node $C_{5,5}$.

Finally, Figure 14 shows the last part of the search. We check that the maximal occurrences continue appropriately to the beginning of the pattern. Two occurrences pass the test and are reported, for example $B_9 \ldots = P_{2\ldots}$, since reading upwards from $Node(8)$ we obtain $P^r_{1\ldots1}$.

Figure 15 depicts the whole algorithm. Occurrences are reported in the format $(k, offset)$, where k is the identifier of the block where the occurrence starts and $offset$ is the distance between the beginning of the occurrence and the end of the block.

If we want to show the text surrounding an occurrence $(k, offset)$, we just go to $LZTrie$ using $Node(k)$ and use the $parent_t$ pointers to obtain the characters of the block in reverse order. If the occurrence spans more than one block, we do the same for blocks $k+1$, $k+2$ and so on until the whole pattern is shown. We also can show larger block numbers as well as blocks $k-1$, $k-2$, etc. in order to show a larger text context around the occurrence. Indeed, we can recover the whole text by repeating this process for $k \in 0 \ldots n$.

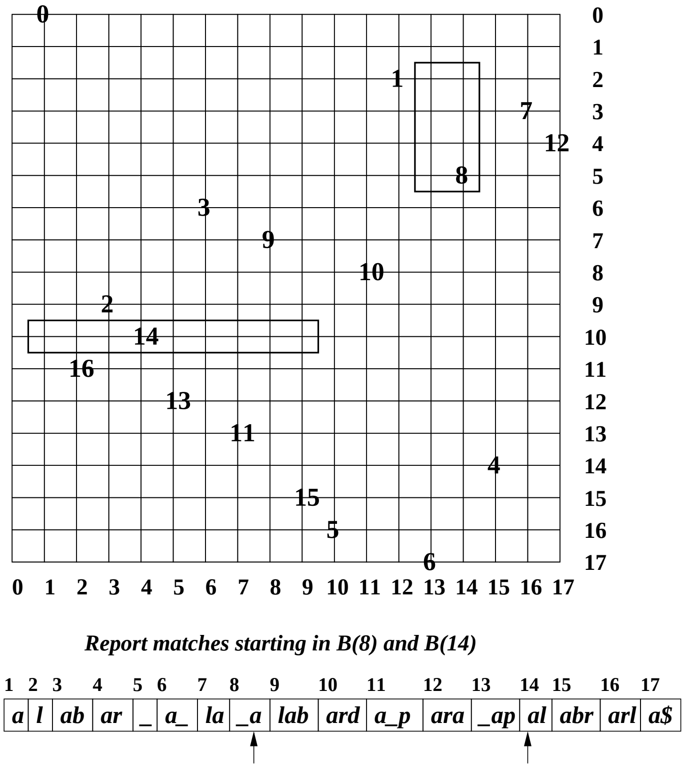


Figure 10: Reporting occurrences of type 2 of $P = ala$ in our running example, third part.

4 A Succinct Index Representation

We show now how the data structures used in the algorithm can be implemented using little space. It should be clear that this is part of our theoretical proposal, which guarantees the complexities we have promised. In Section 6 we will change several decisions in order to obtain a practical implementation.

Let us first consider the tries. Munro and Raman [22] show that it is possible to store a binary tree of N nodes using $2N+o(N)$ bits such that the operations $parent(x)$, $leftchild(x)$, $rightchild(x)$ and $subtreesize(x)$ can be answered in constant time. Munro et al. [23] show that, using the same space, the following operations can also be answered in constant time: $leafrank(x)$ (number of leaves to the left of node x), $leafsize(x)$ (number of leaves in the subtree rooted at x), $leftmost(x)$ and $rightmost(x)$ (leftmost and rightmost leaves in the subtree rooted at x).

In the same paper [23] they show that a trie can be represented using this same structure by representing the alphabet Σ in binary. This trie is able to point to an array of identifiers, so that the identity of each leaf can be known. Moreover, path compressed tries (where unary paths are compressed and a skip value is kept to indicate how many nodes have been compressed) can be represented without any extra space cost, as long as there exists a separate representation of the strings stored readily available to compare the portions of the pattern skipped at the compressed paths.

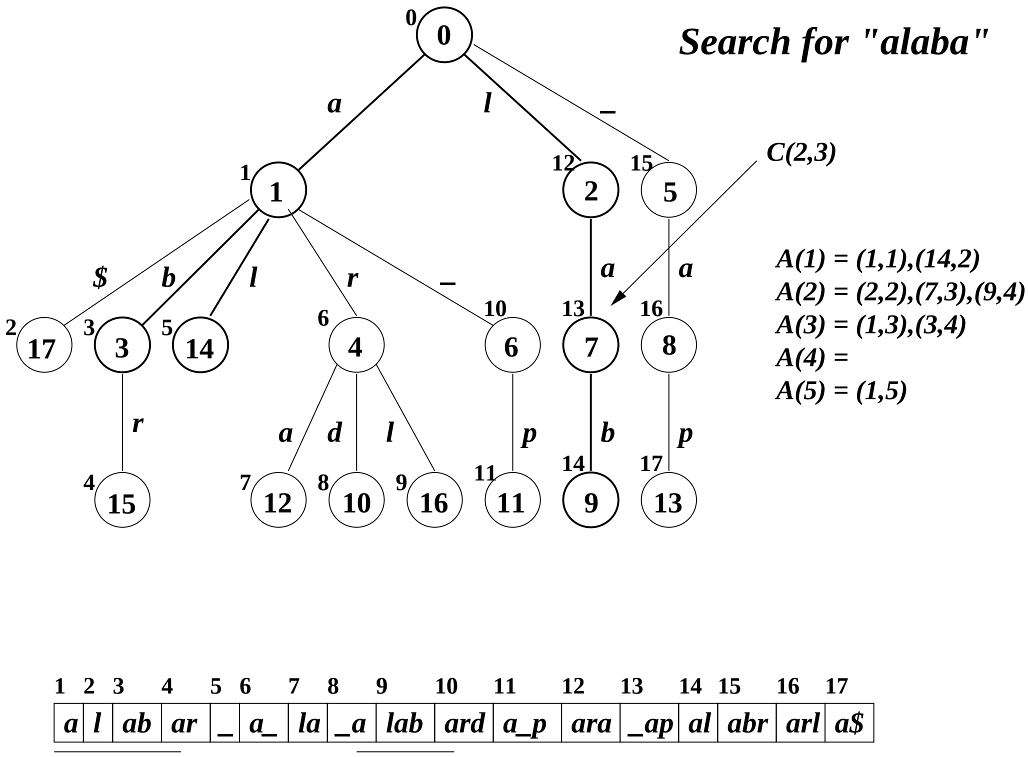


Figure 11: Reporting occurrences of type 3 of $P = alaba$ in our running example, first part.

We use the above representation for $LZTrie$ as follows. We do not use path compression, but rather convert the alphabet to binary and store the $n + 1$ strings corresponding to each block, in binary form, into $LZTrie$. For reasons that are made clear soon, we prefix every binary representation with the bit "1". So every node in the binary $LZTrie$ will have a path of length $1 + \log_2 \sigma$ to its real parent in the original $LZTrie$, creating at most $1 + \log_2 \sigma$ internal nodes. We make sure that all the binary trie nodes that correspond to true nodes in the original $LZTrie$ are leaves in the binary trie. For this sake, we use the extra bit allocated: at every true node that happens to be internal, we add a leaf by the bit 0, while all the other children necessarily descend by the bit 1.

Hence we end up with a binary tree of $n(1 + \log_2 \sigma)$ nodes, which can be represented using $2n(1 + \log_2 \sigma) + o(n \log \sigma)$ bits. The identity associated to each leaf x will be $id_t(x)$. This array of node identifiers is stored in order of increasing rank, which requires $n \log_2 n$ bits, and permits implementing rth_t in constant time.

The operations $parent_t$ and $child_t$ can therefore be implemented in $O(\log \sigma)$ time. The remaining operations, $leftrank(x)$ and $rightrank(x)$, are computed in constant time using $leafrank(leftmost(x))$ and $leafrank(rightmost(x))$, since the number of leaves to the left corresponds to the rank in the original trie.

For $RevTrie$ we have up to n leaves, but there may be up to u internal nodes. We use also the binary string representation and the trick of the extra bit to ensure that every node that represents a block is a leaf. In this trie we do use path compression to ensure that, even after converting

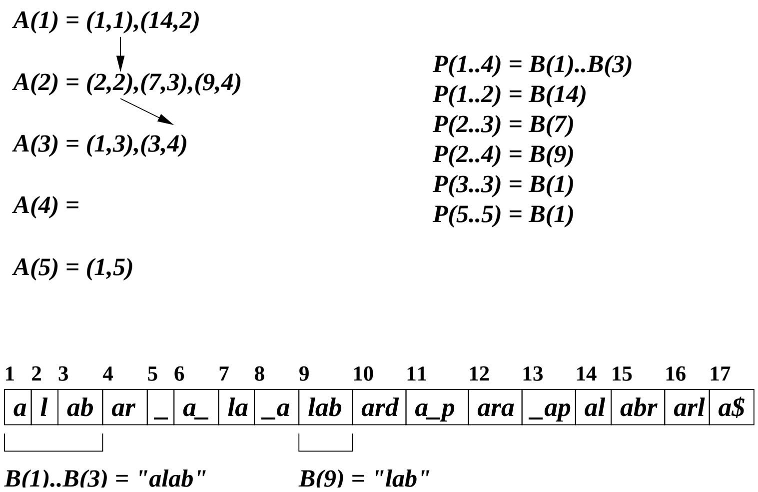


Figure 12: Reporting occurrences of type 3 of $P = alaba$ in our running example, second part.

the alphabet to binary, there are only n nodes to be represented. Hence, all the operations can be implemented using only $2n + o(n)$ bits, plus $n \log_2 n$ bits for the identifiers.

It remains to explain how we store the representation of the strings in the reverse trie, since in order to compress paths one needs the strings readily available elsewhere. Instead of an explicit representation, we use the same $LZTrie$. Assume that we are at a reverse trie y node representing string a, and we have to consider going down to the child node x. To find out which is the string b joining y to x, we obtain, using $Node(rth_r(leftrank(x))$ and $Node(rth_r(rightrank(x))$, two nodes in $LZTrie$. We have to go up from both nodes until we read a^r (string a reversed), and then we continue going up to the parent in $LZTrie$. What we read after a^r is b^r. The process finishes when the characters read from both nodes is different or one reaches the root of $LZTrie$. Note that advancing to a child may require $O(m \log \sigma)$ time in $RevTrie$.

For the $Node$ mapping we simply have a full array of $n \log_2 n$ bits.

Finally, we need to represent the data structure for range searching, $Range$, where we store n block identifiers k (representing the pair $(k, k+1)$). Among the plethora of data structures offering different space-time tradeoffs for range searching [2, 13], we prefer one of minimal space requirement by Chazelle [5]. This structure is a perfect binary tree dividing the points along one coordinate plus a bucketed bitmap for every tree node indicating which points (ranked by the other coordinate) belong to the left child. There are in total $n \log_2 n$ bits in the bucketed bitmaps plus an array of the point identifiers ranked by the first coordinate which represents the leaves of the tree.

This structure permits two dimensional range searching in a grid of n pairs of integers in the range $[0 \ldots n] \times [0 \ldots n]$, answering queries in $O((R+1) \log n)$ time, where R is the number of occurrences reported. A newer technique for bucketed bitmaps [11, 21] needs $N + o(N)$ bits to represent a bitmap of length N, and permits the $rank$ operation (now meaning number of 1's up to a given position) and its inverse in constant time. Using this technique, the structure of Chazelle requires just $n \log_2 n(1 + o(1))$ bits to store all the bitmaps. Moreover, we do not need the information at the leaves, which maps rank (in a coordinate) to block identifiers: as long as we know that the r-th block qualifies in normal (or reverse) lexicographical order, we can use rth_t (or

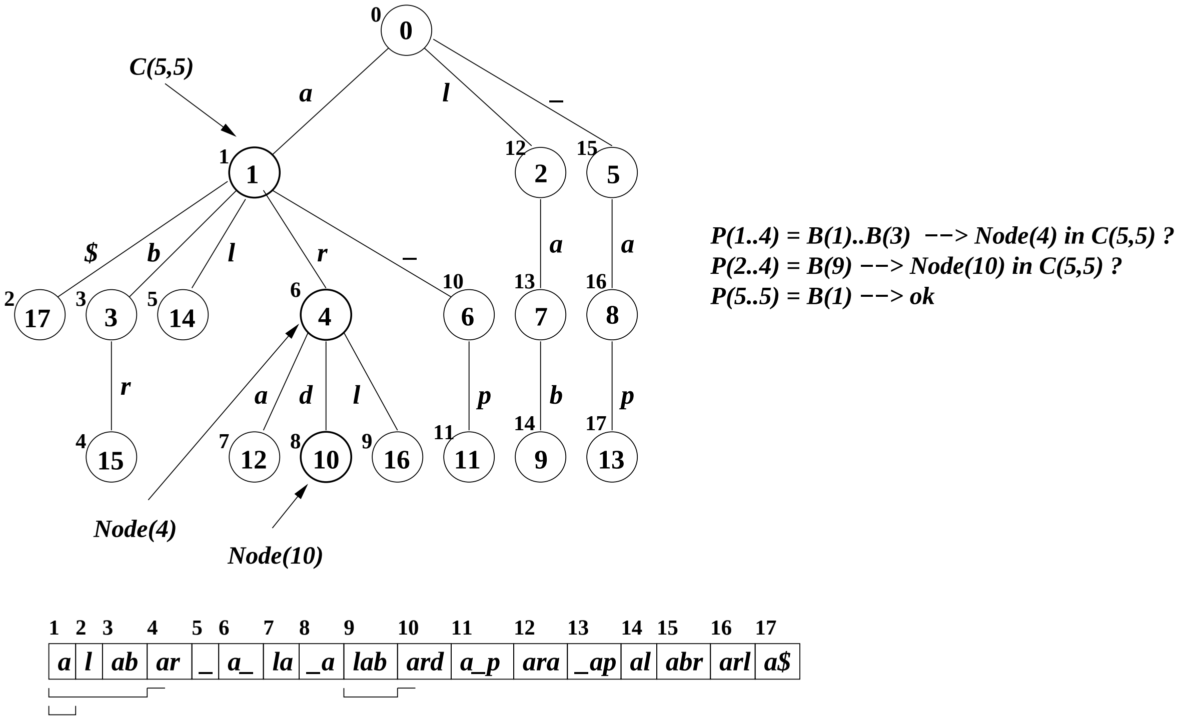


Figure 13: Reporting occurrences of type 3 of $P = alaba$ in our running example, third part.

rth_r) to obtain the identifier $k+1$ (or k).

5 Space and Time Complexity

From the previous section it becomes clear that the total space requirement of our index is $n\lceil\log_2 n\rceil(4+o(1))$ bits. The tries and $Node$ can be built in $O(u\log\sigma)$ time, while *Range* needs $O(n\log n)$ construction time. Since $n\log n = O(u\log\sigma)$ [4], the overall construction time is $O(u\log\sigma)$. Let us now consider the search time of the algorithm.

Finding the blocks that totally contain P requires a search in *RevTrie* of cost $O(m^2\log\sigma)$. Later, we may do an indeterminate amount of work, but for each unit of work we report a distinct occurrence, so we cannot work more than R, the size of the result.

Finding the occurrences that span two blocks requires m searches in $LZTrie$ and m searches in $RevTrie$, for a total cost of $O(m^3\log\sigma)$, as well as m range searches requiring $O(m\log n + R\log n)$ (since every distinct occurrence is reported only once).

Finally, searching for occurrences that span three blocks or more requires m searches in $LZTrie$ (all the $C_{i,j}$ for the same i are obtained with a single search), at a cost of $O(m^2\log\sigma)$. Extending the occurrences costs $O(m^2\log m)$. To see this, consider that, for each unit of work done in the loop of lines 27–29 in Figure 15, we mark one C cell as *used* and never work again on that cell. There are $O(m^2)$ such cells. This means that we make $O(m^2)$ binary searches in the A_i arrays. The cost to sort the m arrays of size m is also $O(m^2\log m)$. The final verifications to the right and to

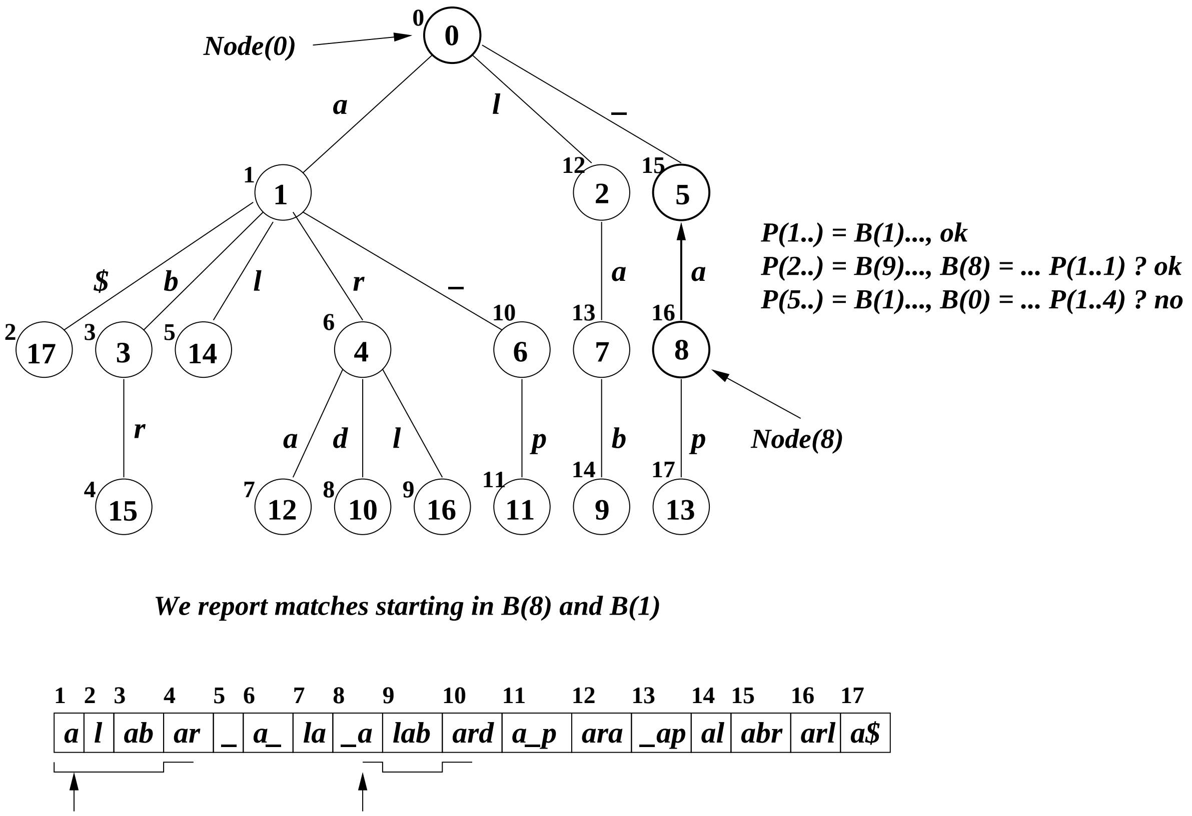


Figure 14: Reporting occurrences of type 3 of $P = alaba$ in our running example, fourth part.

the left cost $O(1)$ and $O(m \log \sigma)$, respectively, and there may be $O(m^2)$ independent verifications. Note that we have not included the time to search the left piece in $RevTrie$, in which case the costs would have raised to $O(m^4 \log \sigma)$. The reason is that, overall, we have to search for every reversed substring of P, which requires $O(m^2)$ moves in $RevTrie$, for a total cost of $O(m^3 \log \sigma)$.

Hence the total search cost to report the R occurrences of pattern $P_{1\ldots m}$ is $O(m^3 \log \sigma + (m + R) \log n)$. If we consider the alphabet size as constant then the algorithm is $O(m^3 + (m+R) \log n)$. The existence problem can be solved in $O(m^3 \log \sigma + m \log n)$ time (note that we can disregard in this case blocks totally containing P, since these occurrences extend others of the other two types). Finally, we can uncompress and show the text of length L surrounding any occurrence reported in $O(L \log \sigma)$ time, and uncompress the whole text $T_{1\ldots u}$ in $O(u \log \sigma)$ time.

Chazelle [5] permits several space-time tradeoffs in his data structure. In particular, by paying $O\left(\frac{1}{\varepsilon} n \log n\right)$ space, reporting time can be reduced to $O(\log^{\varepsilon} n)$. If we pay for this space complexity, then our search time becomes $O(m^2 \log(m\sigma) + m \log n + R \log^{\varepsilon} n)$.

```
Search (P_{1...m}, LZTrie, RevTrie, Node, Range)
                /* Lying inside a single block */
    x ← search for P^r in RevTrie
    For r ∈ leftrank_r(x) ... rightrank_r(x) Do
        y ← Node(rth_r(r))
        For z in the subtree rooted at y Do
            Report (id_t(z), m + depth(y) − depth(z))
                /* Spanning two blocks */
    For i ∈ 1 ... m − 1 Do
        x ← search for P^r_{1...i} in RevTrie
        y ← search for P_{i+1...m} in LZTrie
        Search for [leftrank_r(x) ... rightrank_r(x)]
                ×[leftrank_t(y) ... rightrank_t(y)] in Range
        For (k, k + 1) in the result of this search Do Report (k, i)
                /* Spanning three or more blocks */
    For i ∈ 1 ... m Do
        x ← root node of LZTrie
        A_i ← ∅
        For j ∈ i ... m Do
            If x ≠ null Then x ← child_t(x, P_j)
            C_{i,j} ← x
            used_{i,j} ← FALSE
            If x ≠ null Then A_i ← A_i ∪ (id_t(x), j)
    For j ∈ 1 ... m Do
        For i ∈ i ... j Do
            If C_{i,j} ≠ null AND used_{i,j} = FALSE Then
                S_0 ← id_t(C_{i,j})
                S ← S_0 − 1, r ← j − 1
                While (S + 1, r') ∈ A_{r+1} Do /* always exists the 1st time */
                    used_{r+1,r'} ← TRUE
                    r ← r', S ← S + 1
                span ← S − S_0 + 1
                If i > 1 Then span ← span + 1
                If r < m Then span ← span + 1
                If span ≥ 3 Then
                    If C_{r+1,m} = null OR
                        leftrank_t(C_{r+1,m}) ≤ leftrank_t(Node(S + 1)) ≤ rightrank_t(C_{r+1,m}) Then
                        x ← Node(S_0 − 1), i' ← i − 1
                        While i' > 0 AND parent_t(x) ≠ null
                                    AND x = child(parent_t(x), P_{i'}) Do
                            x ← parent_t(x), i' ← i' − 1
                        If i' = 0 Then Report (S_0 − 1, i − 1)
```

Figure 15: The search algorithm. The value $depth(y) - depth(z)$ is determined on the fly since we traverse the whole subtree of z.

6 Implementation: Theory and Practice

We describe in this section the implementation of our index. We review the data structures used one by one, as there are interesting lessons on theory versus practice for each of them.

To demonstrate the results in practice, we have chosen two different text collections. The first, ZIFF, contains 83.37 megabytes (Mb) obtained from the "ZIFF-2" disk of the TREC-3 collection [10]. The second, DNA, contains 51.48 Mb from *GenBank* (Homo Sapiens DNA, `http://www.ncbi.nlm.nih.gov`), with lines cut every 60 characters. We use the whole collections as well as incremental subsets of them.

Our tests have been run on a Pentium IV processor at 2 GHz, 512 Mb of RAM and 512 kilobytes (Kb) of cache, running Linux SuSE 7.3. We compiled the code with `gcc 2.95.3` using optimization option `-O9`. Times were obtained using 10 repetitions for indexing and 10,000 for searching, obtaining percentual errors below 1% with 95% confidence. As we work only in main memory, we only consider CPU times.

The search patterns were obtained by pruning text lines to their first m characters. In the case of DNA we avoided patterns with 5 or more 'N' characters, which represents an unknown character and is not searchable. For ZIFF we avoided lines containing tags and non-visible characters such as '&'.

6.1 Bucketed Bitmaps

One of the lowest level data structures is the bucketed bitmap. This is an array of bits able to answer the query $rank(i)$, which is the number of 1's before position i. In [11, 21] they show how to use $\ell + o(\ell)$ bits to represent a bitmap of length ℓ, implementing $rank$ and its inverse ($select$) in constant time. It turns out that we need just $rank$, which is the easiest.

According to [21], one should divide the bitstream into superblocks of size $s = \log_2^2 \ell$, each of which should be divided into blocks of size $b = (\log_2 \ell)/2$. For each superblock one stores the number of 1's present before the superblock. The same is done for the blocks, but counting only from the beginning of its superblock. Hence we need $\ell / \log_2 \ell$ bits for the superblocks and $4\ell \log_2 \log_2 \ell / \log_2 \ell$ bits for the blocks. Finally, a table is built with precomputed answers for all the possible block contents and i values, which takes $2^b b \log_2 b = \frac{1}{2}\sqrt{\ell} \log_2 \ell (\log_2 \log_2 \ell - 1)$. For example, if $\ell = 2^{26}$ (64 megabits), the extra space turns out to be 81.40% over that of the bitstream itself. Note that, since we use bit streams to represent parentheses, $\ell = 64$ megabits will mean $n = 32$ mega-Ziv-Lempel-blocks, which roughly corresponds to 320 Mb of text. This shows that the "sublinear" part decreases rather slowly.

Note that the table of precomputed answers does no more than "popcounting", that is, counting the number of 1's in a bit mask. There are well known folklore solutions to do this without a precomputed table. Probably the best is

```
bx = x - ((x>>1) & 0x77777777) - ((x>>2) & 0x33333333) - ((x>>3) & 0x11111111)
popcount = ((bx + (bx>>4)) & 0x0F0F0F0F) % 0xFF
```

where a computer word of 32 bits is assumed. The idea is to add up the 1's by consecutive pairs, so as to obtain 2-bit sums, then add up pairs of 2-bit sums to obtain 4-bit sums, and so on. For a computer word of w bits, the procedure requires $O(\log w)$ operations. In the RAM model

$w = O(\log \ell)$, so the cost becomes $O(\log\log \ell)$ instead of constant. For 32 bits, the above solution requires 13 operations, all on a single register. This is usually faster than a single access to RAM memory.

A reasonable principle in our balance between theory and practice is to consider $O(\log\log \ell)$ as good as a constant. Indeed, $\log_2\log_2 \ell \le 5$ for $\ell \le 2^{32}$, which is usually much larger than any size we will handle in main memory. Moreover, $\log_2\log_2 \ell \le 6$ for $\ell \le 2^{64}$, which is 16 million terabits.

Since we replace the precomputed table by popcounting, we can set $b = w$. Using superblocks of size s gives a space requirement of $(\ell/s)\log_2 \ell + (\ell/b)\log_2 s$ bits. Optimizing we get $s = b\log_2 \ell = w\log_2 \ell$. Once we obtain the number of bits to represent a value in $[0, s-1]$ (that is, $\lceil \log_2 s \rceil$, we redefine s as $2^{\lceil \log_2 s \rceil}$ to make the best use of the bits in the blocks.

Our above example with 64 megabits, using $w = 32$, would be as follows: $s = 32 \times 26 = 832$, hence we need $\lceil \log_2 s \rceil = 10$ bits per block counter, and redefine $s = 1024$. We need only 33.79% of extra space. Figure 16 (right) shows, among other things, the experimental space overhead of the bitmaps in our examples. It ranges between 33.30% and 33.60% as expected.

Superblock and block counters are stored in separate arrays of bits, so that each number uses the fixed number of bits we assigned to superblocks and blocks. The bitstream, on the other hand, is stored as a sequence of computer words.

In order to obtain $rank(i)$, we add (1) the value of the superblock counter number $i >> \lceil \log_2 s \rceil$; (2) the value of the block counter number $i >> \log_2 w$; (3) the popcount of the word number $i >> \log_2 w$ of the bitstream, after removing all but the first x bits of it, where x is given by the $\log_2 w$ lowest bits of i. Since we use powers of 2, we can resort to bit shifting instead of the slower multiplication and division.

6.2 Balanced Parentheses

The proposals in [22, 23] to handle trees and tries in succint space build on top of a succint representation of a sequence of balanced parentheses. As we follow the same path, we study now in depth a practical implementation, keeping in mind that our goal is to represent a general tree in the obvious way (ancestors enclose their descendants).

The solution proposed in [22] to store a sequence of p parentheses uses $o(p)$ extra space and permits executing the following operations in constant time: $findopen(i)$ and $findclose(i)$ find the position of the opening(closing) parenthesis that matches closing(opening) parenthesis at position i; $excess(i)$ gives the excess of opening parentheses over closing parentheses up to position i; and $enclose(i)$ gives the opening position of the closest parentheses pair that encloses opening parenthesis i.

In practice, the solution of [22] is overwhelmingly complicated, involving a complex structure that is replicated at three levels (big, small, and tiny blocks). Which is worse, the extra "sublinear" space is extremely large in practice. The authors mention that the sequence should have more than 20 million bits before the excess becomes less than 100%. However, the situation is much worse. We made the exercise of adding up all the sublinear-size data structures for our 64 megabit example, and it turned out that the extra space is 15 times p. Moreover, we did not count a precomputed table whose size is $2^{4(\log_2\log_2 p)^2} 8(\log_2\log_2 p)^2 (2 + 2\log_2\log_2\log_2 p) \approx 2^{111}$ bits.

It may be true that this overhead is necessary to ensure constant time in the required operations. But it is also clear that we cannot go ahead with this approach in practice. Hence we design a simpler scheme that guarantees $O(\log \log p)$ average time and (almost) guarantees bounded extra space. We do reuse some of the ideas of [22], but we combine them with practical considerations.

6.2.1 General Scheme

We represent the balanced sequence of parentheses as a bucketed bitmap, where 0 represents an opening parenthesis and 1 a closing parenthesis. This gives us the $rank(i)$ operation (Section 6.1), which makes our $excess(i)$ operation extremely simple, as it is the number of 0's minus 1's. This is a simple function of $rank(i)$, namely $excess(i) = i - 2 \times rank(i)$. We use the $excess(i)$ function several times in which follows.

Let us for now focus on $findopen(i)$ and $findclose(i)$, as $excess(i)$ will then come easily. Given an opening (for $findclose()$) or closing (for $findopen()$) parenthesis, what we need is to know the position of its matching parenthesis.

Our aim is to explicitly store the position of the matching parenthesis only for large subtrees (recall that our parentheses represent general trees). That is, if a subtree has less than $b/2$ nodes (b parentheses) we will find its matching parenthesis by brute force, that is, looking at the bits that follow or precede it until we find the first with the same excess. We will not work more than $O(b)$ at this. For larger subtrees, we will store the answer directly in a hash table. If there are no unary nodes, then there cannot be more than p/b large subtrees.

This is not enough if we attempt to work $O(\log \log p)$, since we would have to store p/b numbers of $\log_2 p$ bits, for a total space requirement of $O(p \log p / \log \log p)$. We instead define three types of parentheses: "close" parentheses are those whose matching parentheses are at distance at most b; "near", between $b+1$ and s; and "far", farther than s bits away. The hash table for far parentheses uses $\log p$ bits to store the values, while that for near parentheses uses just $\log s$ bits (we do not store the absolute position of the matching parenthesis but its distance from the argument). Close parentheses are solved by brute force.

6.2.2 Hashing

Implementing the hashing scheme is not as trivial as it might seem. We exclude perfect hashing, whose implementation is not practical for our very large data sets[2]. We opt for a closed hashing scheme with table size at least 1.8 times the number of elements, which is a good tradeoff between space and search time (expected number of accesses per search is 2.25)[3]. Hashing is done by multiplying the key by a large constant prime, and rehashing by adding another such prime (so we suffer from clustering, but address computation is cheaper). The table size is a power of 2 to avoid computing modulus. This turned out to be much faster at a small price in space.

The interesting problem we have to face is how to handle collisions *without storing the keys* (as the keys are absolute positions requiring $\log_2 p$ bits). In general there would be no solution to this problem: there is no way to distinguish to which of two colliding keys does a given value correspond. However, in our particular case, an elegant solution exists.

[2] We found good implementations, by the authors, of the best current schemes, and they could not handle more than a few thousand keys. See `http://www.amk.ca/python/code/perfect-hash.html`.

[3] We are considering the cost of an unsuccessful search, $1/(1-\alpha)$, $\alpha = 1/1.8$, because, as it should be clear from the next paragraphs, we have to consider all the elements that collide before deciding which is ours.

Imagine that we search for a given parenthesis position in our hash table and recover a set of possible answers (corresponding to other keys that have collided with our key). Each such answer gives a possible distance to our matching parenthesis. It is not hard to discard those candidates to matching positions whose excess is different from that of our key. However, there could be still several candidates with the correct excess. If we know that our key is in the table, then our solution is in the answer and it can only be the closest matching position with correct excess. This gives us a simple solution, with the only restriction that we have to be sure that our key is in the set (otherwise we could get confused with the answer for a colliding key).

We first try to find the matching parenthesis by brute force up to distance b. If we fail, then the parenthesis is either near or far. We search the hash table of near parentheses, assuming that the parenthesis is near. If it is, we will find the correct answer. If it is not, no colliding answer can fool us, because every position between our key and the answer has excess larger than that of our key (because the sequence is balanced), and the answers of the near table can give us only near positions. Hence we will detect that our parenthesis is far because no suitable answer will be found in the set of answers. In that case, we will search the far table, where the answer surely is.

We remark that, although we have to consider "sets" of colliding answers for correctness, in practice we expect typically at most two colliding answers.

In principle, we need indeed two near and two far tables: one for opening and one for closing parenthesis, to implement $findopen(i)$ and $findclose(i)$, respectively. Soon we will show a slight modification to this idea.

6.2.3 Brute Force Search

Finally, let us look in depth at the solution for small subtrees (not in the hash tables). We do not really search for their matching parenthesis bit by bit. We rather process the following or preceding bits by chunks of k bits. This is done as follows. For every different k-bit stream, we precompute its excess, and also, for every $j \in 1 \ldots k$, the first position of the k-bit stream where the excess becomes $-j$, if any.

To solve $findclose(i)$, we isolate the b bits following position i and traverse them by chunks: We ask if the excess -1 is reached in the first chunk. If it is, the position where this happens is that of the closing parenthesis. Otherwise, let e_1 be the excess of the first chunk. We then consider the second chunk, looking for excess $-1 - e_1$. If it is reached inside the second chunk, we have the answer, otherwise, if e_2 is the excess of the second chunk, we continue looking for excess $-1 - e_1 - e_2$, and so on. If we exhaust the b bits without a solution, then the answer is not close.

Solving $findopen(i)$ is almost identical, precomputing tables that look for positive instead of negative excesses, and considering the sequences right to left.

6.2.4 Solving $enclose(i)$

The operation $enclose(i)$ can indeed be solved using $findopen()$ as follows. If we are the first child of our parent, we can detect that because the parenthesis at position $i-1$ is an opening parenthesis, and in this case this is the opening position of the parent (enclosing pair). Otherwise, we can find the previous sibling as $findopen(i-1)$. We traverse the sequence of siblings backward until we find the parent.

There is no complexity guarantee for this operation except the arity of the parent. In our application this will be σ in the worst case. In practice, this turned out to be rather slow and the operation turned out to be needed in many places of the overall scheme, ruining the overall performance.

Hence we preferred to directly store the enclosing parenthesis of each opening parenthesis. This is the last parenthesis before i whose excess is one less than the excess of i. It turns out that the hashing scheme described above fits perfectly well, and collisions can be handled the same way. The brute force search for close parentheses is also very similar to that of $findopen()$. Moreover, $findopen()$ itself is not used elsewehere in our application, so we do not need to store information on closing parentheses.

The only drawback of this approach that there will be much more near and far parentheses, as enclosing parentheses are farther away than matching parentheses. The resulting overhead is not negligible but the whole search process becomes much faster. A way to limit the number of near and far parentheses is to see that a subtree of size at most b (s) will have all close answers inside, but its root can have a near (far) answer. In practice we are including one extra level of the tree in the hash tables, which can give us at most $\sigma - 2$ times more extra space than for matching parentheses (the first child has always a close answer, as well as, almost always, the next sibling of a small subtree).

6.2.5 Evaluation

With the above approach we solve any of the operations working at most $O(b/k)$ plus two searches on hash tables, which are $O(1)$ on average. Assuming there are no unary nodes, the space is at most $1.8 \times ((\sigma-1)p/s)\log_2 p + 1.8 \times ((\sigma-1)p/b)\log_2 s + 2^k(2k+1)\log_2 k$ bits, where the first term is for the far parentheses, the second for the near parentheses, and the third for the precomputed tables. The value $\sigma - 1$ comes from 1 (for $findclose()$) plus $\sigma - 2$ (for $enclose()$), the factor 2 from the guarantee of $2p/s$ or $2p/b$ large trees, and the 1.8 from the hashing load factor.

The optimum is $s = b\log_2 p$. If $b = \log_2 p$ and $k = \log_2 p/\log_2\log_2 p$, then $s = \log_2^2 p$. Our search cost remains $O(\log\log p)$ and the space requirement is sublinear and reasonable. In our example of 64 megabits, and assuming $\sigma = 4$, we have an overhead of 2.28 times the stream size, plus a constant number of 1248 bits (recall that this is an upper bound, it is much better on average). Using hash tables that are powers of 2 may drive this extra space up to 4.56 in the worst case.

In practice k can be made larger. For obvious practical reasons we have chosen to fix $k = 8$, so we advance by bytes. Our constant size tables take 4.25 Kb, which admits good caching. We fix $b = w$, in our case $b = 32$. This means that we first process the first 4 bytes, one by one, and then go to the hash tables if necessary. We keep $s = b\log_2 p$. Reconsidering our example, we need at most 1.86 times the bitstream size plus 4.25 Kb (these tables are the same no matter how many bitstreams we handle, so they can be considered as part of the program size).

These figures are much better than 15 times the bitstream size. Recall, however, that they come from a simplified analysis and that they assume there are no unary nodes.

Figure 16 shows the space overheads incurred in our example tests. On the left we have plotted the percentage of "near" and "far" parentheses both in *LZTrie* and *RevTrie* ($parent(i)$ is not used on $RevTrie$). On the right we have plotted the overall space overhead to represent balanced parentheses, including those attributed to the bucketed bitmaps. As we can see, only 2% to 7%

of the parentheses have to be stored in the "near" hash table, and 0.04% to 0.6% are stored in the "far" tables. The percentages seem to stabilize for large texts. The overhead due to these hash tables is, for *LZTrie* 1.5 to 3.6 times the space of the parentheses themselves, and 0.5 to 0.8 for *RevTrie*. The reason for the difference among the tries is the lack of *enclose*() information on *RevTrie*. The reason for the fluctuations in the figures is that we require the hash table size to be a power of two to avoid computing modulus. We could change this decision to remove the fluctuations, but we would pay much more in terms of search time, and these fluctuations hardly influence the overall index size (see later).

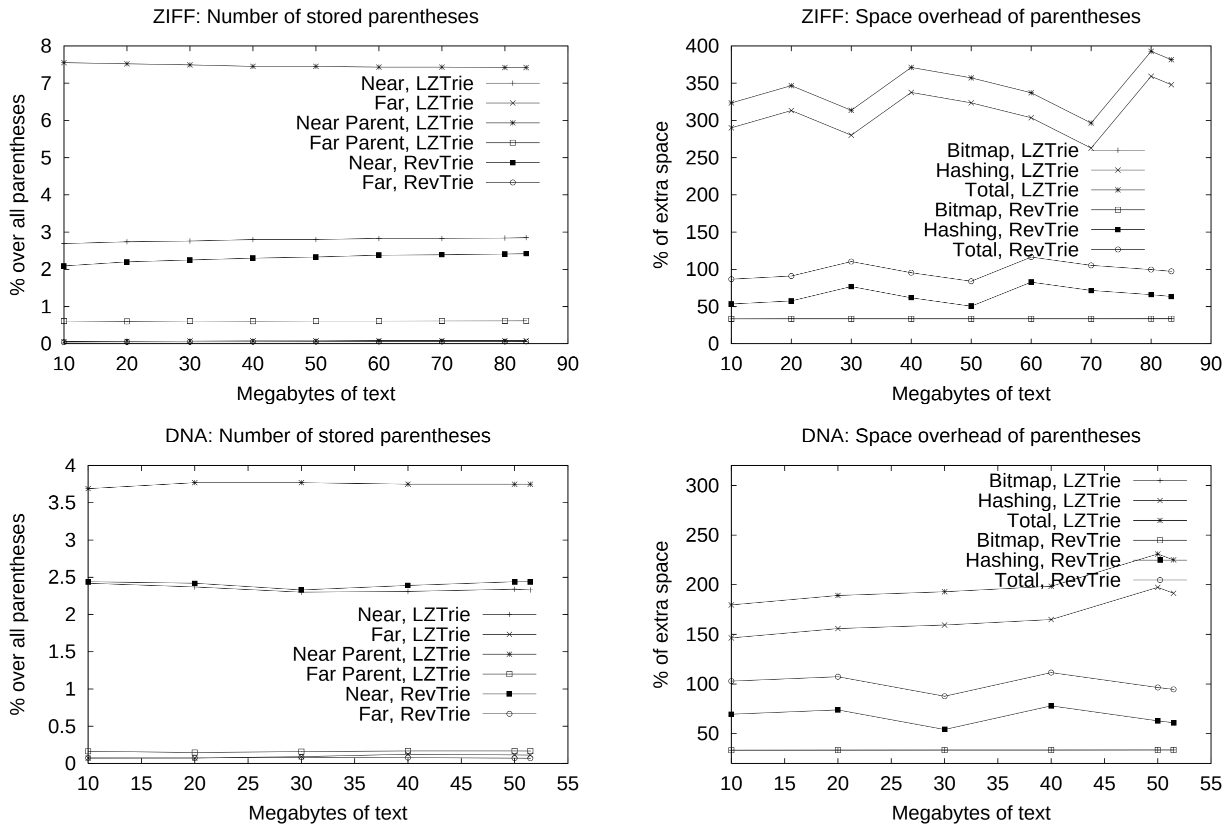


Figure 16: Different aspects of the space overhead to store balanced parentheses.

6.3 LZTrie

We again choose to change the theoretical approach of [22], although we again reuse some of their ideas.

6.3.1 Structure

Instead of converting our alphabet to binary and representing the trie as a binary tree and this in turn as a sequence of parentheses of maximum arity 2, we choose to directly represent the trie in its general tree form, as a sequence of parentheses. The main consequence is that, by converting the alphabet to binary, we would pay $O(\log \sigma)$ for any $child(i, a)$ operation, while with a representation as a general tree we could pay $O(\sigma)$, assuming we search linearly for the proper child a. In practice, however, only the highest nodes of the trie have a significant arity, while most of them will have much less than $\log_2 \sigma$. On the other hand, the direct implementation as a general tree is much simpler and requires less space. To alleviate the arity problem, the answers for the (at most σ) children of the root are precomputed.

The *LZTrie* structure contains a sequence of parentheses representing the trie structure, a sequence *lets* of characters that label each edge of the trie, in preorder, and a sequence *ids* of block identifiers, also in preorder. Given a node (represented as the position i of its opening parenthesis), its position in the sequence of letters/identifiers is easily obtained: it corresponds to the number of opening parentheses (0's) before position i, that is, $i - rank(i)$. Hence the letter by which node i descends from its parent and its identifier are easily obtained.

6.3.2 Implementing the Operations

To compute $child(i, a)$, the child of node i by letter a, we examine the children of i until we find (or not) one that descends by a (we can stop if we find a letter larger than a). The first child is $j = i + 1$ and the others are obtained as $j = findclose(j) + 1$. The children finish when j is a closing parenthesis. To go to the parent node we simply use $parent(i) = enclose(i)$.

Figure 17 shows the number of invocations to $findclose()$ per call to $child(i, a)$. It shows that in practice the cost of this operation in the case of DNA is around 2. For ZIFF, on the other hand, it is around 10–12. It is much higher, however, for short patterns (because for them most of the searches occur in the top of the trie, where arities are larger). For short patterns the cost of the trie search is negligible compared to the overall search cost, as will be clear later.

Note that for very short DNA patterns ($m = 5$), having the first level of the trie preprocessed has a great influence on the overall time. Note also that there is a slight increase as the text grows, as expected.

The other functions we have to provide are easily implementable. First, $subtreesize(i)$, the number of nodes of the subtree rooted at i, is simply half the number of parentheses enclosed by node i, $subtreesize(i) = (findclose(i) - i + 1)/2$. Second, $depth(i)$, the depth in the tree of node i, is exactly $excess(i)$. Third, $leftrank(i)$ and $rightrank(i)$, the first and last lexicographical positions of strings below node i, are simply $leftrank(i) = i - rank(i)$ (as node i represents the smallest string in the subtree) and $rightrank(i) = j - rank(j) - 1$, where $j = findclose(i)$. Fourth, $rth(pos)$, the block identifier of the pos-th string in the trie, is just $ids(pos)$. Fifth, $ancestor(i, j)$[4] tells whether node i is an ancestor of j, and is just $ancestor(i, j) = i \leq j \leq findclose(i)$. Finally, the letter and identifier of node i can be easily rewritten as $letter(i) = lets(leftrank(i))$ and $id(i) = ids(leftrank(i))$. We also implement simple functions that permit traversing the trie in depth first search order (not caring about the letters), which is useful to report whole subtrees (occurrences of type 1).

[4] A new operation we have included for convenience, as seen later.

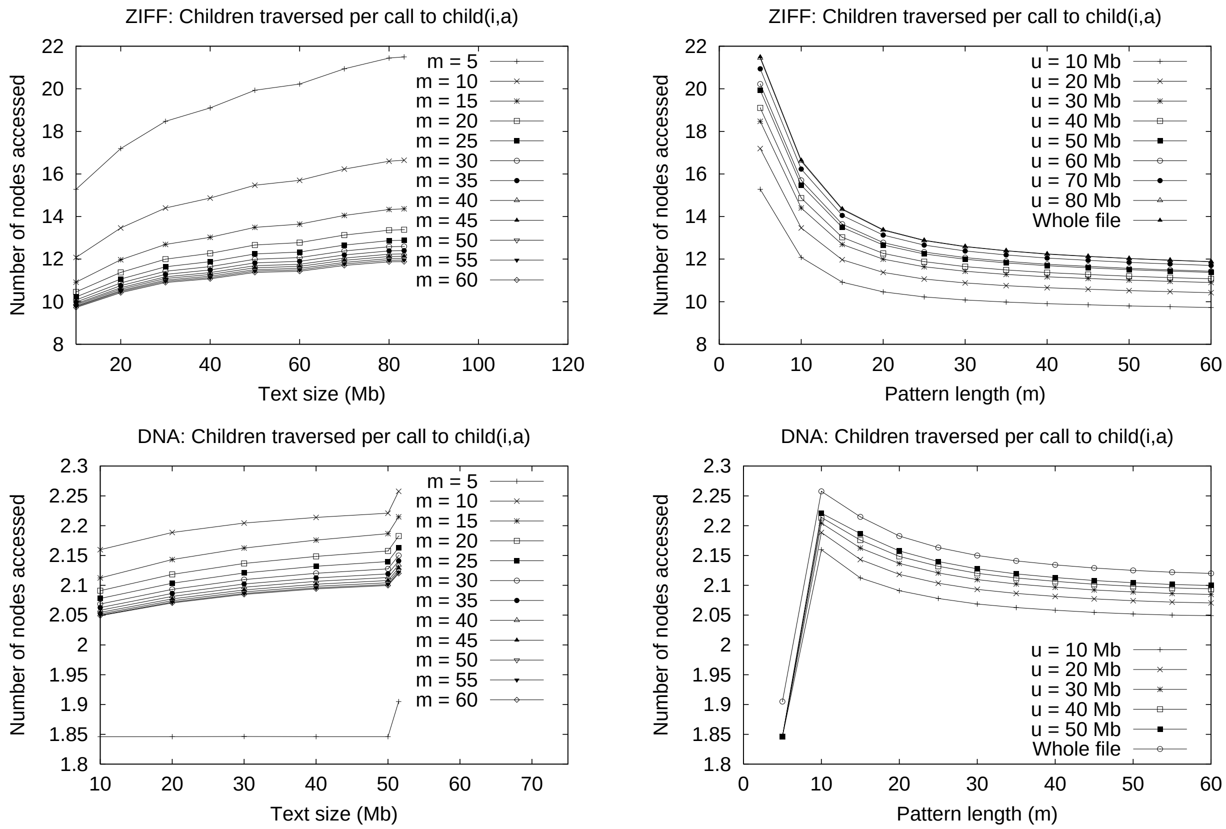


Figure 17: Number of $findclose()$ invocations to find the proper child in the $child(i, a)$ operation.

6.3.3 Construction

LZTrie is built as follows. We traverse the text and at the same time build a normal trie of the strings represented by Ziv-Lempel blocks. At step k, we read the text that follows and step down this trie until we cannot continue. At this point we create a new block (assigning it next block number k), go to the root again, and go on with step $k+1$ reading the rest of the text. The process finishes when the last block finishes with the text terminator "\$".

Figure 18 (left) shows the construction times for *LZTrie*. We have identified four steps: (i) creation of normal trie, where the text is read and the normal trie (with pointers) is built; (ii) representation of this trie using parentheses, which involves simply traversing it and writing down bits when a new tree is started/finished; (iii) freeing the normal trie, which is essential to limit the overall construction RAM space; and (iv) creation of the compressed trie representation, which means creating a balanced parentheses data structure using the bit stream that represents the trie, and creating the arrays of letters and identifiers.

As expected, the most time-consuming process is by far the creation of the trie. This shows that any improvement on this aspect will result in a significant decrease of the overall construction time. It is also clear that the times are slightly superlinear, although the algorithms are clearly linear, both in the average and worst case. The reason is most probably the reduced locality of reference as larger tries are built. At their maximum sizes, the overall construction speed is about 3 Mb/sec for DNA and 2 Mb/sec for ZIFF.

In fact, freeing the trie was initially time consuming too, as we had to free all the individual nodes. As there were few different sizes to work with, we decided to create our own memory manager handling objects of fixed size, using a different manager for each different object size. This manager allocates nodes in blocks, and freeing them is much faster than the naive approach. Some programming languages (like Modula-2) or runtime support systems provide independent "heaps", that is, memory areas that are allocated and freed once and that support allocation of memory inside them. This kind of heap is what we would really need.

6.3.4 Extra Space

Figure 18 (right) shows the extra space needed by *LZTrie*. The larger value is the space (as a fraction of the text size) needed by the normal trie. The smaller value is the space after we have it in compressed form. It can be seen that the space requirement drops as the text grows, which is a consequence of having a number of nodes equal to the number of blocks, which grows as $n = O(u/\log u)$. For large texts, the extra space needed to build the normal trie becomes 1.7 to 2.0 times the text size. This space is freed as soon as we have enough information to build the compressed representation.

6.4 Node

There is nothing special in the implementation of the $Node$ data structure. $Node$ is just a bit stream with as many bits as necessary to represent n nodes of $LZTrie$ (that is, positions 0 to $2n-1$). It is built as the inverse of the array ids of $LZTrie$.

6.5 RevTrie

6.5.1 Structure

The reverse trie has several similarities with $LZTrie$, but also some important differences. The trie is also represented by a sequence of balanced parentheses and a sequence of block identifiers, but this time (1) the edge between two nodes can be labeled by a string, which is not represented; (2) we remove unary nodes that have no block identifier, but still non-unary nodes without block identifiers remain and are represented (these will be called *empty* nodes); (3) we do not implement the $parent$ operation.

A first question is which is the number of nodes of $RevTrie$, since we have the same n nodes of $LZTrie$ plus some empty nodes. As can be seen in Figure 19 (left), the extra $RevTrie$ nodes stabilize around 3.4% over $LZTrie$ nodes on ZIFF and around 2.2% on DNA. So the price for these empty nodes, which are extremely convenient for search purposes, is minimal. On the right we can see that the number of trie nodes per text character decreases, as expected from a Ziv-Lempel parsing. What may be not so clearly expected is that $RevTrie$ nodes decrease at the same rate, posing a constant overhead over $LZTrie$ nodes.

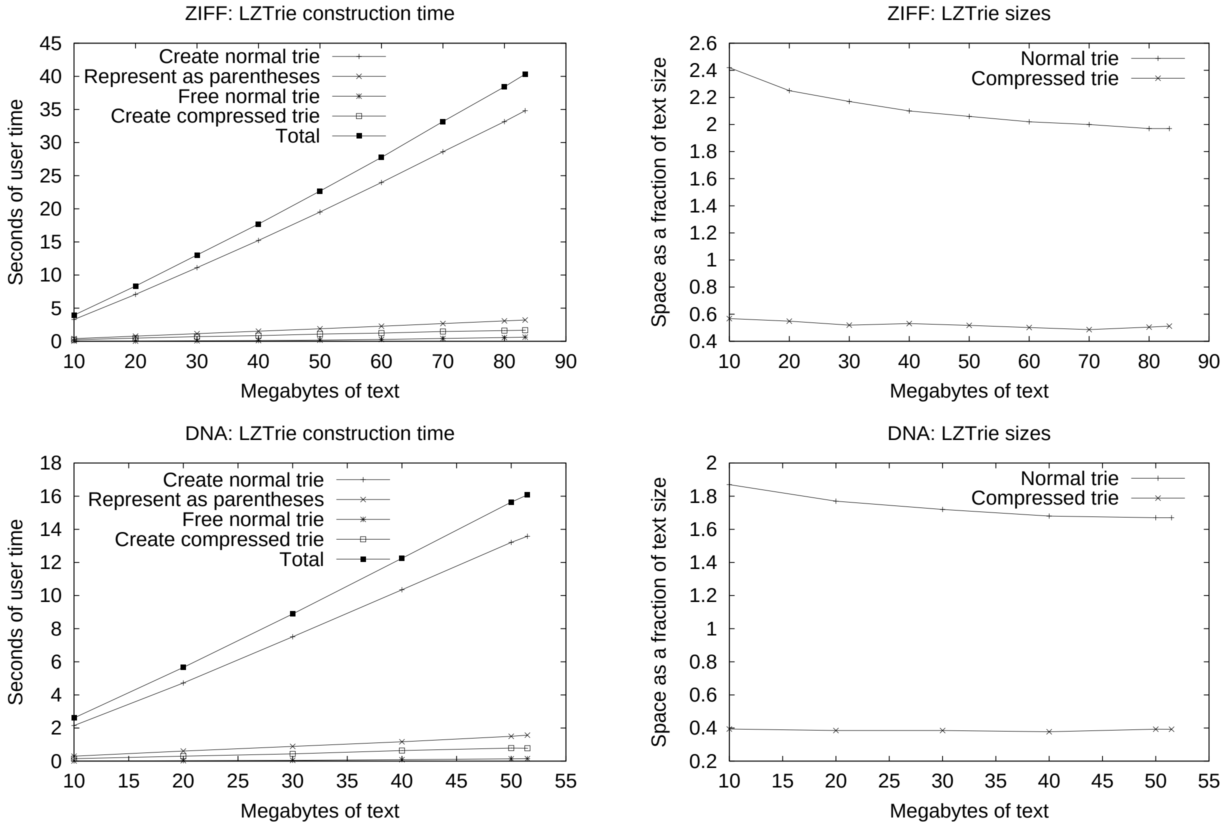


Figure 18: Different aspects of the construction of the *LZTrie* data structure.

As we have to leave the space of empty nodes in the sequence of identifiers, it is convenient to give them the same identifier of their lexicographically smallest non-empty descendant (leaves are never empty).

6.5.2 Implementing the Operations

The only complex problem is how to implement $child(i, a)$, because (1) edges are labeled by full strings, and (2) we do not have any representation of these strings. The problem is solved thanks to the connections to $LZTrie$. Let us say that node i represents string x. For each child j of i, we map j to $LZTrie$ using $j_t = Node(rth_r(rank(j)))$. From j_t we go via $parent()$ operations until we traverse upwards string x. One step further in this path tells us which is the character c by which j descends from i. If $c = a$, then j is the correct child.

If and when we determine that j is the correct child of i by letter a, we have to determine which is the string that joins i to j. This string can be obtained by going up more steps in $LZTrie$, but we need to know where to stop. Since, by construction of $RevTrie$, the identifier of j is that of the lexicographically smallest string in the subtree (be j an empty node or not), we only have to compute $rightrank(j)$ to have the smallest and largest strings in the subtree. We map also

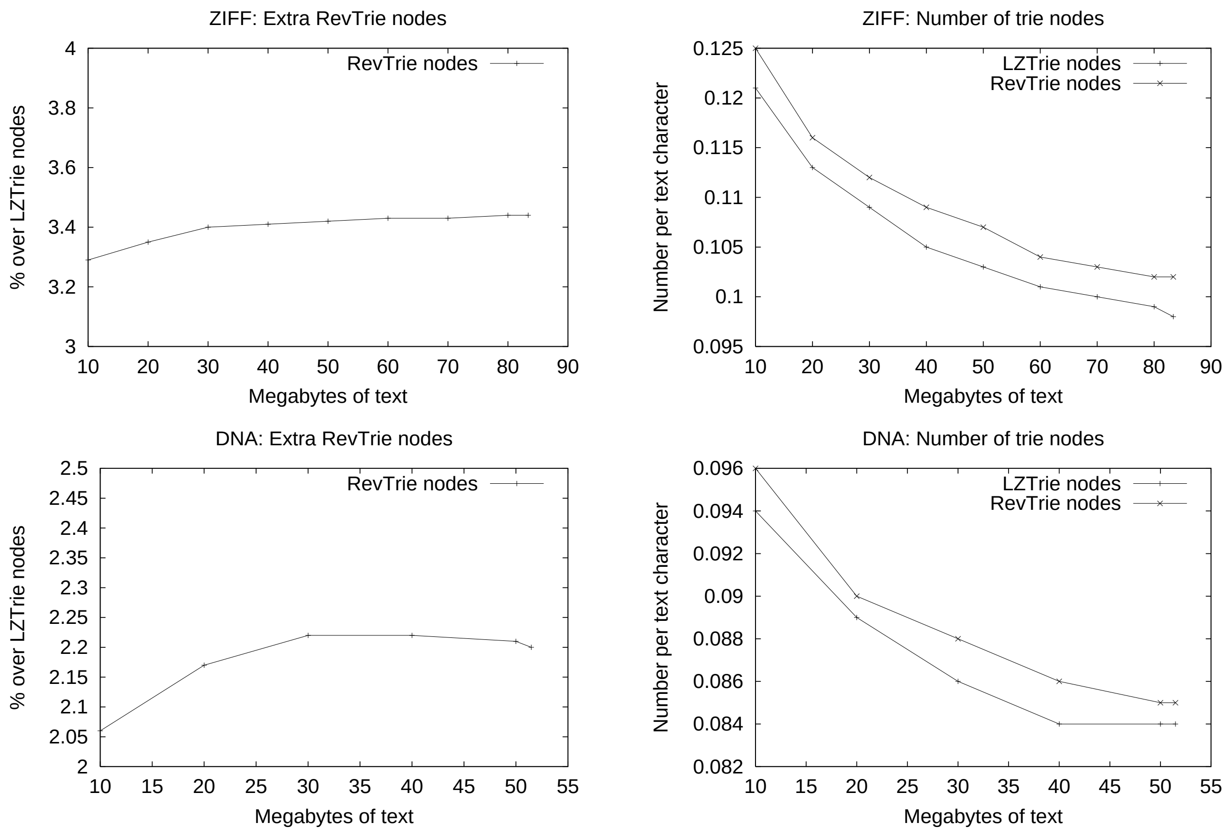


Figure 19: Number of trie nodes in *LZTrie* and *RevTrie* data structures.

$j'_t = Node(rth_t(rightrank(j)))$ to $LZTrie$, and go up string xa. At this point we go up from both nodes in $LZTrie$. As soon as their characters differ, we have read the string that joins i to j. This process is done interactively with the calling method in order to solve the next calls to $child(child(i, a), b)$ fast. Only when we finally arrive at j we have to rescan the set of children, go up their paths in $LZTrie$, and so on. Nevertheless, the process is tedious and slow, so we seek to limit it as much as possible.

6.5.3 Construction

The construction of *RevTrie* is done as follows. We traverse $LZTrie$ in a depth-first-search manner, generating each string stored in $LZTrie$ in constant time, and then inserting it into a normal trie of reversed strings. For simplicity, we have not compressed unary paths in the normal trie. When we finish, we traverse the trie and represent it using a sequence of parentheses and block identifiers, and at the same time remove empty unary nodes.

Figure 20 (left) shows the construction times for *RevTrie*. We have identified the same four steps as for $LZTrie$. Again, the most time-consuming process is by far the creation of the trie. Although a bit better than the construction costs for $LZTrie$, these times dominate the overall construction cost. Again we can see a slightly superlinear increase due to caching.

6.5.4 Extra Space

What has worsened a lot is the extra space needed to hold the normal trie. Figure 20 (right) shows that we need around 4 times the text size for ZIFF, and 2.5 times for DNA. This shows that the extra space to represent unary empty nodes is around 77%–85% for ZIFF and 35%–50% for DNA. After we compress unary paths, the compressed representation becomes even smaller than that of *LZTrie*, as we do not store the *lets* array. The space of the normal trie is freed as soon as we have enough information to build its compressed representation, but it influences the maximum amount of main memory we need across all the indexing process. In particular, using path compression at the time of the construction of the normal trie would greatly reduce the overall space requirement.

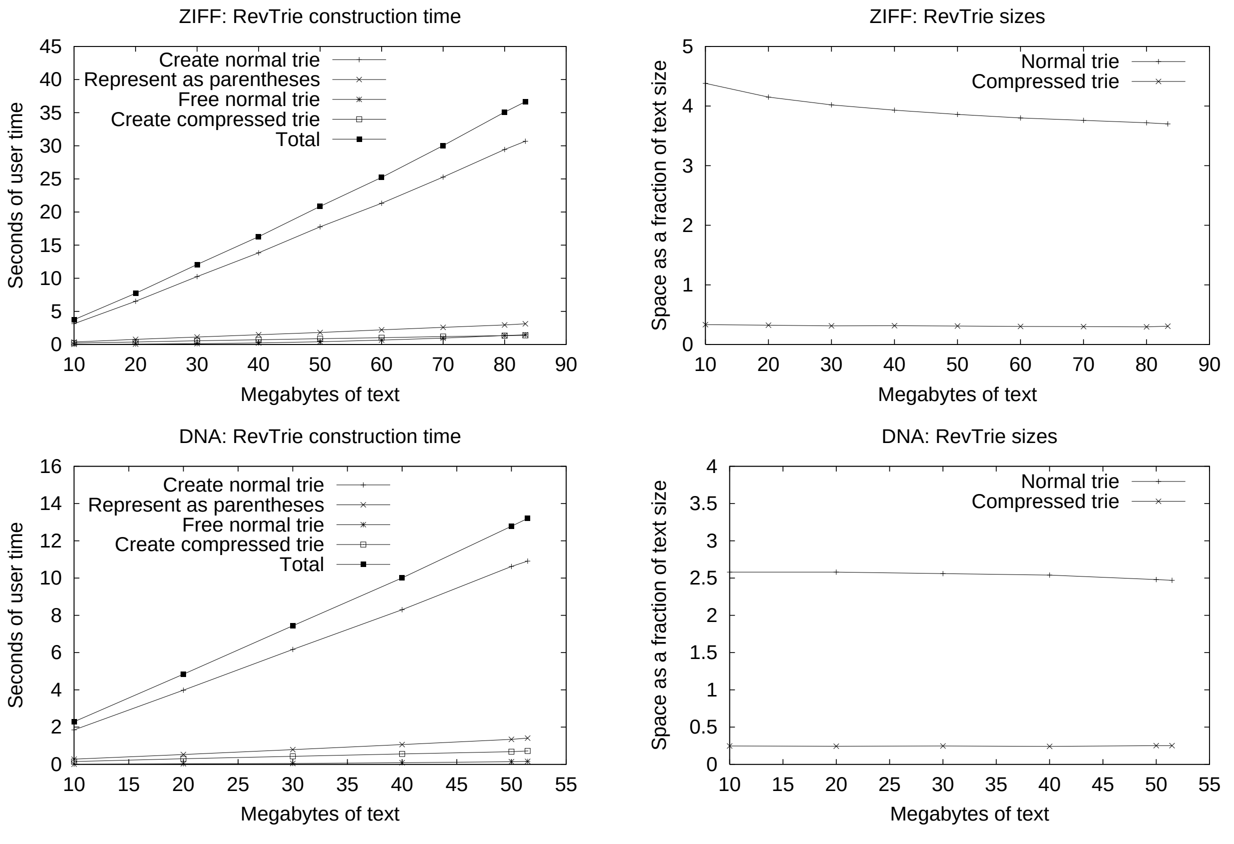


Figure 20: Different aspects of the construction of the *RevTrie* data structure.

6.6 Range versus RNode

In the beginning, we implemented the *Range* data structure pretty much as described in [5] (Section 4). The only interesting improvement was that we noticed that, if the points (x, y) represented a bijection, then it would be possible to know the initial position of the right subtree (in the first level there would be exactly $2^{\lceil \log_2 n \rceil}/2$ zeros, and so on). This would save us some $rank()$ computations and would permit an easy construction algorithm. Hence we completed the mapping with fake values so as to make it a bijection (originally it mapped $LZTrie$ node labeled $k+1$ to $RevTrie$ node labeled k, so it was not a bijection because there are extra empty $RevTrie$ nodes). The price in space was rather moderate and the structure was faster.

When we tested it for searching, however, it turned out that, by far, the most time-consuming part of the search was step 2, which is the one related to *Range*. All our attempts to speed up the execution failed. We then realized that *Range* occupied $n' \log n'$ bits, where $n' \approx 1.03n$ was the number of $RevTrie$ nodes (indeed more, since we needed about 35% extra space for the bitmaps in order to implement $rank()$), and that with this space we could instead store a reverse $Node$ data structure, $RNode$. $RNode$ maps block identifiers to their (nonempty) nodes in $RevTrie$.

With $RNode$ we could solve quite decently the same problem addressed by *Range*, as follows. Say that the search for $P^r_{1\ldots i}$ in $RevTrie$ leads us to node i_r and the search for $P_{i+1\ldots m}$ in $LZTrie$ leads us to node i_t (if any of the two nodes does not exist we know immediately that this partition of P produces no matches[5]). Both for i_t and i_r, we can use $rank$ and $rightrank$ to determine the ranges in the ids arrays where the relevant blocks lie. Then we have two choices:

(a) For each block $k+1$ in the portion of ids corresponding to $LZTrie$, ask whether $ancestor(i_r, RNode(k))$. If so, report block k.

(b) For each block k in the portion of ids corresponding to $RevTrie$, ask whether $ancestor(i_t, Node(k+1))$. If so, report block k.

Since it is easy to determine which will require less work, we choose the best among both choices. Without $RNode$ we would have been forced to use always option (b), which is very bad when i is small because the area is too large. On average, we expect the area of $RevTrie$ to be of size n/σ^i and that of $LZTrie$ of size n/σ^{m-i}. So without $RNode$ we would have to work $n\sum_{i=1}^{m-1} 1/\sigma^i \approx n/\sigma$ time, which is huge. With $RNode$ we work on average $n(\sum_{i=1}^{m/2} 1/\sigma^{m-i} + \sum_{i=m/2+1}^{m-1} 1/\sigma^i) \approx n/\sigma^{m/2}$, which is much better.

An immediate concern is that, under the new scheme, in order to intersect two sets, we are able to work in time proportional to the smallest size, and this is inferior to the *Range* complexity, which works $O((R+1)\log n)$ independently of the set sizes. It could happen that we intersect large sets whose final result is small, and hence *Range* behaves much better. What happens in practice is that R has a lot to do with the sizes of the sets. Large pairs of sets produce a large R and hence *Range* is slow too (note that the worst cases occur for small m values). On average, if the candidate sizes are A and B, we expect the result to be of size $R = AB/n$. This means that on average *Range* works $n \log n \sum_{i=2}^{m} 1/\sigma^m = O(mn \log n/\sigma^m)$. Comparing against the cost of the $RNode$ based approach we have that our new approach is inferior for $m > 2\log_\sigma(m \log n)$. For example, if $n = 32$ mega-blocks of DNA, *Range* is better for $m > 8$.

However, as we will see later, the cost of this search becomes much less important for that length. A way to express this is as follows. If we add up the search cost for every pattern length between 1 and M, we get asymptotically $\sigma n \log n$ using *Range* and $n/\sqrt{\sigma}$ using $RNode$. So if we

[5] If, in $RevTrie$, we are in the middle of an edge, we can safely traverse the edge and consider the child as the correct solution.

consider that every length is equally probable, $RNode$ is superior by far, as it wins where the costs are higher. In practice, we found that the version based on $RNode$ took half the time of $Range$ on short patterns (for example searching all the English words of a dictionary on ZIFF) and 2/3 the time on long patterns, no matter the length of the text.

Finally, having $RNode$ at hand simplifies and speeds up the search at other moments, as we will describe soon. As an additional benefit, the index is about 8% smaller when we replace $Range$ by $RNode$.

6.7 Building the Index

Construction of the overall index proceeds as follows.

1. We build the normal Ziv-Lempel trie.
2. We represent it with parentheses, letters and identifiers. The array of identifiers is not yet bit-packed but an unpacked array of numbers.
3. We free the normal trie.
4. We build $LZTrie$ using the array of parentheses, letters and identifiers. The unpacked array of identifiers is still maintained.
5. We free the text, as it is not anymore necessary.
6. We build the mapping array as the inverse of the identifiers array, not yet bit-packed.
7. We free the array of unpacked identifiers (we kept it until building the mapping array because accessing it is faster in unpacked form).
8. We create the bit-packed $Node$ data structure and free the unpacked map.
9. We build the normal reverse trie.
10. We represent it with parentheses and identifiers. Again the array of identifiers is unpacked.
11. We free the reverse trie.
12. We build $RevTrie$ using the array of parentheses and identifiers. The unpacked array of identifiers is still maintained.
13. We create the unpacked reverse mapping array as the inverse of the array of identifiers, in unpacked form.
14. We free the unpacked array of identifiers.
15. We create the bit-packed $RNode$ data structure and free the unpacked map.

This sequence is designed to minimize the maximum amount of main memory required to index. The maximum is usually reached after step 10. Note that we soon free the text and never need it again. Figure 21 (left) shows the maximum amount of main memory required at indexing time. As it can be seen, the percentage of extra space required drops as the text grows, influenced by the smaller amount of trie nodes. For ZIFF we need from 4.8 to 5.8 times the text size, while for DNA this drops to 3.4 to 3.7. As a comparison, the construction of a plain suffix array without any extra data structure requires 5 times the text size. The difference, of course, is that, after we build the index, we are left with a very succint representation, while a normal suffix array needs those 5 times the text size forever.

Figure 21 (right) shows construction costs. It becomes clear that the construction of the tries dominate the overall construction cost, and that this is slightly superlinear. In the case of ZIFF, for 10 Mb we build the index at 0.82 sec/Mb, while for the whole 83.37 Mb text this has grown to 0.97 sec/Mb. For DNA the figures are 0.53 sec/Mb on 10 Mb and 0.61 sec/Mb on 51.48 Mb. In general we can speak of a construction speed of 1–2 Mb/sec, which is much better than the construction costs of, for example, suffix arrays.

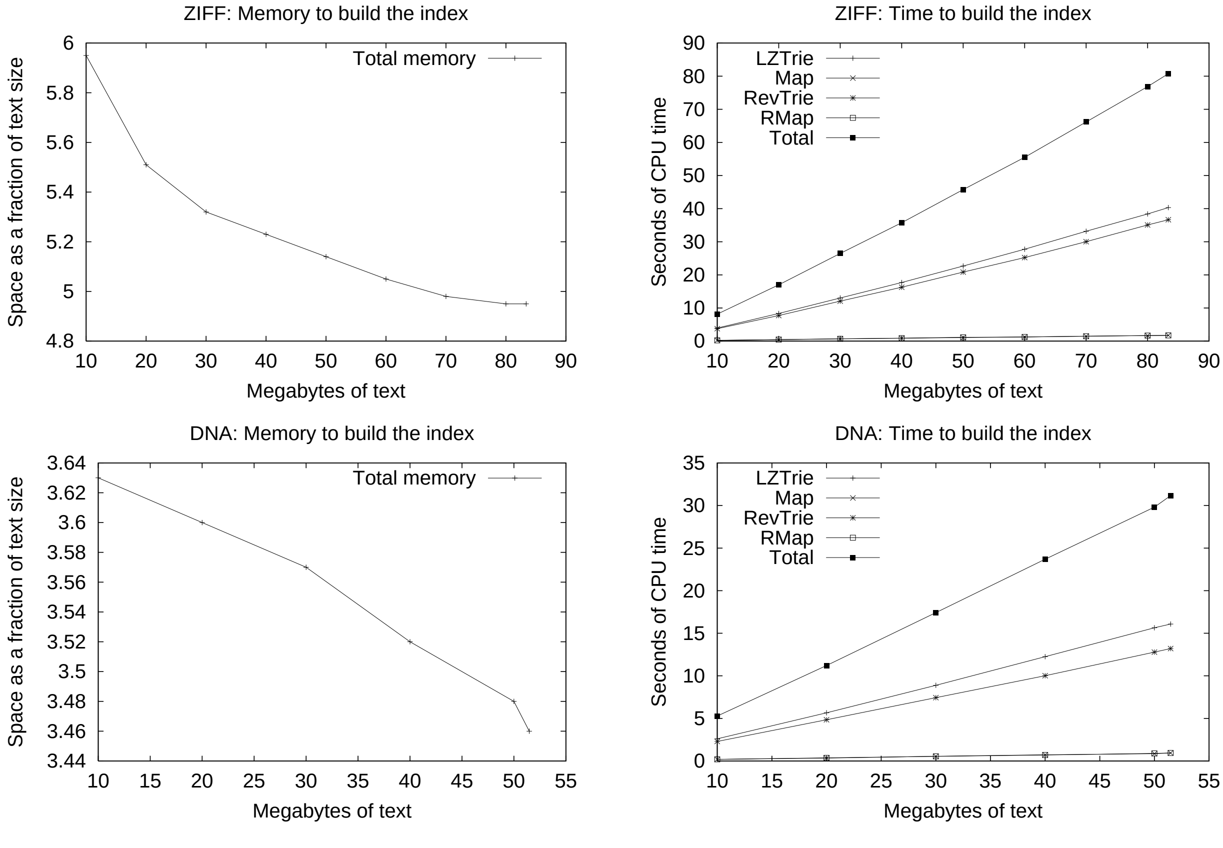


Figure 21: Different aspects of the construction of the whole index.

Finally, Figure 22 (left) shows the space requirement of the finished indexes. The oscillations of the parentheses representation are hardly noticeably. The space requirement drops as the text sizes grow. For ZIFF, we require from 1.6 to 1.4 times the text size (and this includes what we need to reproduce the text), while for DNA the figure stays around 1.2.

The compression ratios obtained are related to the compressibility of the text. To show this more clearly and to test our predictions of using 4 times the space needed by the compressed text, we show the space of the final index as a fraction of the file size after we compress it with Unix's *Compress* program (a LZW compressor), as well as as a fraction of $n \log n$, being n the number of Ziv-Lempel blocks. The latter accounts for a pure LZW compression scheme where the text is not cut into buffers, and should give a better estimate of the Ziv-Lempel compressibility of the text, while the *Compress* program is more useful as a control value.

As can be seen in Figure 22 (right), Compress obtains compression ratios which are below the theoretical optimum, as it cuts the text in chunks (typically of 64 Kb) and compresses each chunk separately. Our theoretical computation, on the other hand, shows the invariant we were looking for: independent of the type of the text, our index takes about 5 times the size of the Ziv-Lempel compressed text (without chunking). As our prediction was 4 times the text size, the rest accounts for terms considered sublinear: the arrays of letters and of parentheses, hash tables, etc.

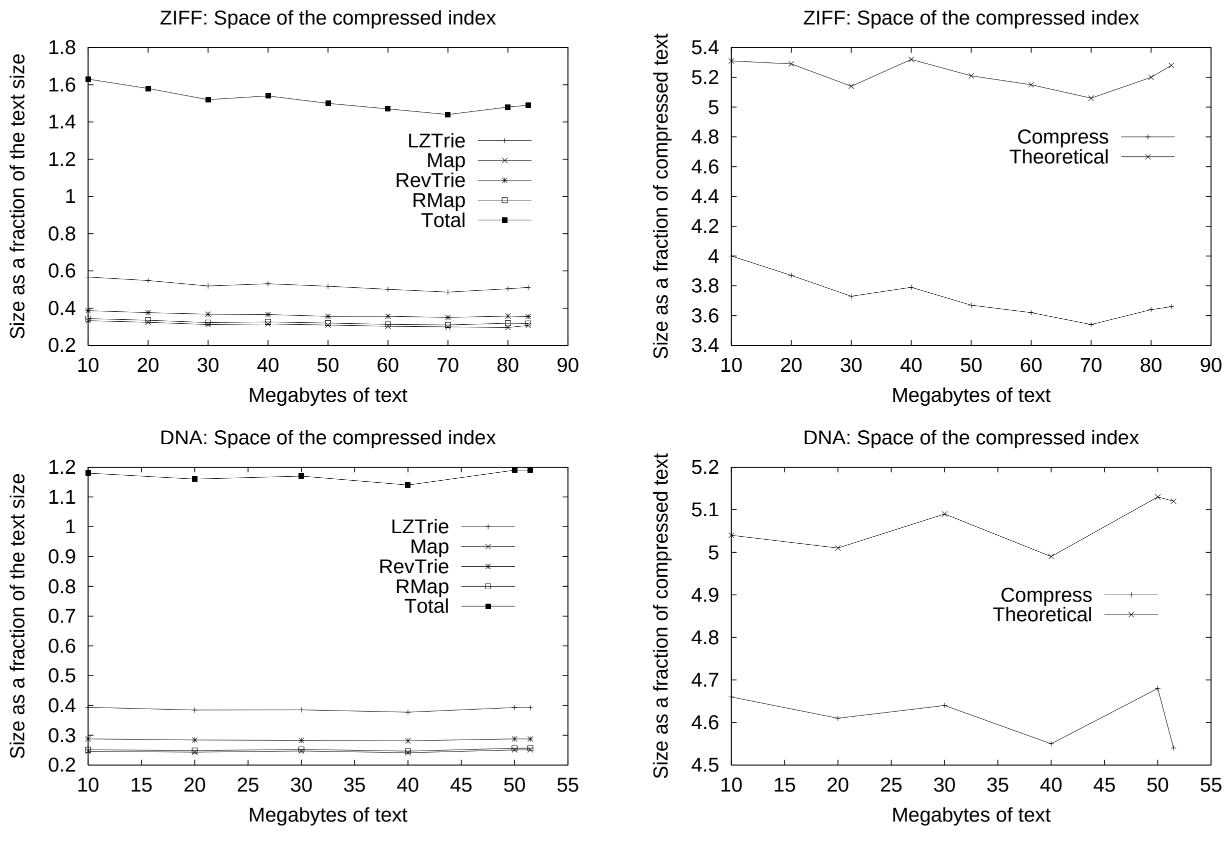


Figure 22: Sizes of our compressed indexes.

6.8 Searching

The search process is divided into five steps. These are not exactly as described before. We first search for all the pattern substrings, then for all their reverses, and then report each type of occurrences, 1 to 3.

To analyze the empirical behavior of the search process, let us call ℓ the average length of a LZ78 phrase (that is, $\ell = u/n$ if u is measured in bytes and n in blocks). It holds $\ell = \Omega(\log u)$. The larger ℓ, the more compressible the text is, as it can be represented in $(u/\ell)\lceil \log_2 u/\ell \rceil$ bytes. Except for very special texts, most phrases have length ℓ, and also $\ell = \Theta(\log u)$. In our experiments (Figure 19) we obtain very good approximations of the form $\ell = 6.16 + 0.22 \ln u$ on ZIFF and $\ell = 8.74 + 0.18 \ln u$ on DNA.

6.8.1 Building Matrix $C_{i,j}$

We search for every pattern substring $P_{i...j}$ using $LZTrie$, and obtain the matrix $C_{i,j}$ of the nodes corresponding to each substring, if any. We also obtain a matrix of block identifiers $Cid_{i,j}$ corresponding to each node $C_{i,j}$. Matrix $Cid_{i,j}$ is necessary at several points, most evidently to report occurrences of type 3. As explained, to search for the $O(m^2)$ strings we traverse $O(m^2)$ edges in $LZTrie$, since the node for $P_{i,j+1}$ must be a child of the node for $P_{i,j}$. However, we can work less because once $P_{i...j}$ is not present in $LZTrie$ we know that $P_{i...j+k}$ is not present either for any k.

On average, we do not expect to fill all the $m^2/2$ cells. Only strings up to length ℓ appear in $LZTrie$, and hence we expect to fill $O(m\ell)$ cells of $C_{i,j}$. This is confirmed in Figure 23, where we have drawn the number of cells really filled and an $O(m \log u)$ pattern is rather clear.

Filling each cell involves a $child(i, a)$ operation, whose cost is proportional to the average arity of the trie. Most of these operations are done close to the root, where the arity is close to its maximum σ. Hence on average we work $O(\sigma m \min(m, \log u))$ time to fill $C_{i,j}$. This is obviously a simplification if we regard Figure 17. However, it works well because, for large m, the average arity traversed stabilizes (around 11 for ZIFF and 2 for DNA). For small m, on the other hand, the times are negligible. The difference in arity (11 to 2) is explained if we define σ as the inverse of the probability of two random characters being equal for each text. This gives $\sigma = 4.8$ on DNA (recall that there are newlines and N's) and $\sigma = 20.0$ on ZIFF.

Let us now focus on real experimental times. Figure 24 shows the time to fill matrix $C_{i,j}$ (and $Cid_{i,j}$). The expected $O(m \log u)$ pattern is still visible, although a bit hidden by variance. We have used least squares with the model $t = a + b\sigma m \ln u$ and obtained

$$\begin{aligned} \text{ZIFF} &= -165.336 + 0.244\sigma m \ln u \quad \mu secs \\ \text{DNA} &= -60.640 + 0.342\sigma m \ln u \quad \mu secs \end{aligned}$$

with a percentual error[6] below 7% in both cases. (By μsecs we mean microseconds.)

[6] We denote by this the measure $100 \times \left(\sum_{i=1}^{N} |y_i - x_i|/y_i\right)/N$.

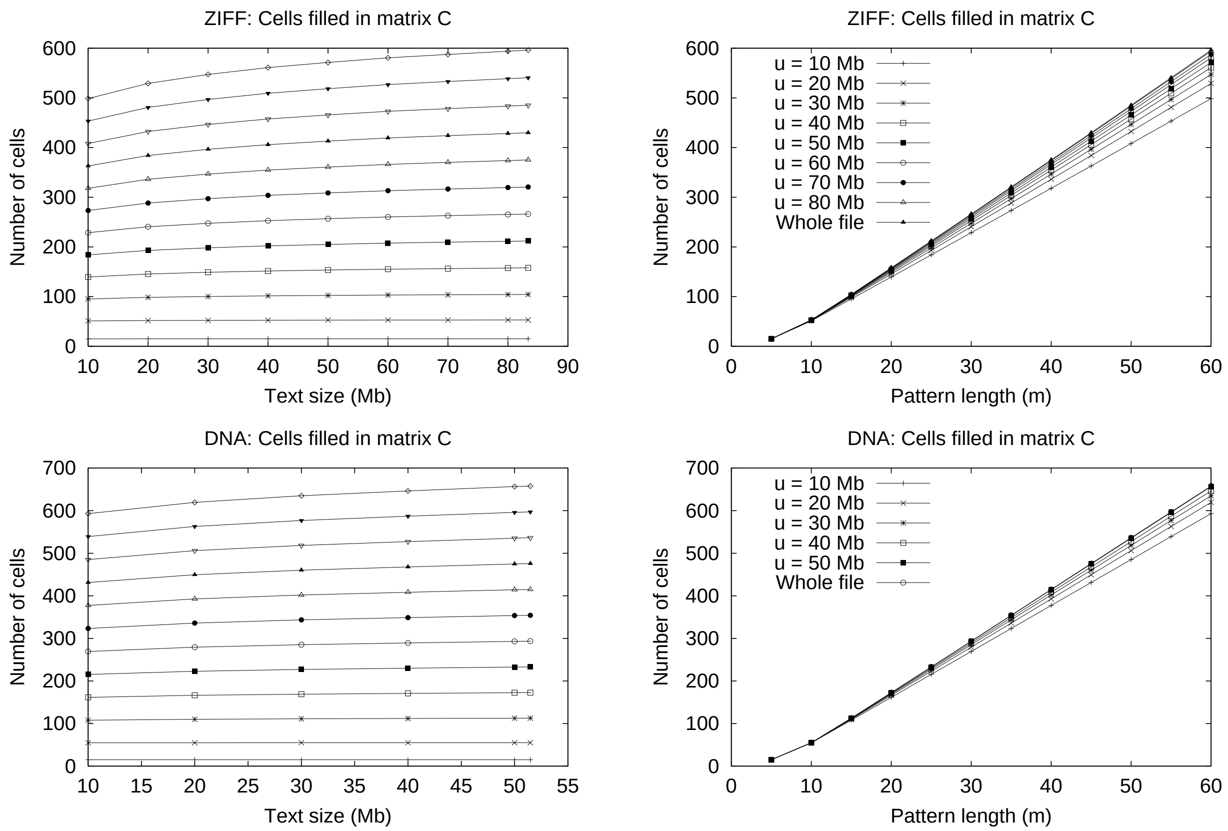


Figure 23: Number of cells actually filled in matrix $C_{i,j}$. On the left, the lines represent, from lower to upper, $m = 5$, $m = 10$, and so on until $m = 60$.

6.8.2 Building Vector B_j

The second step searches for every reversed pattern prefix, $P^r_{1...j}$, in $RevTrie$, and stores it in an array B_j. This is necessary to report occurrences of type 1 and 2. Since searching in $RevTrie$ is much slower than on $LZTrie$, we seek to reduce this work as much as possible. The results already obtained in Cid are useful. If we look for $P^r_{1...j}$ and $P_{1...j}$ exists in $LZTrie$ (that is, $C_{1,j}$ is not null), then $RNode(Cid_{1,j})$ directly gives us the corresponding node in $RevTrie$. Otherwise, $P^r_{1...j}$ corresponds to an empty node or to a position in a string between two nodes, and cannot be directly found with $LZTrie$. Still, we can reduce the search cost as follows. Let i be the minimum value such that $C_{i,j}$ is defined. Then $RNode(Cid_{i,j})$ is the lowest nonempty ancestor of the node we are looking for. We can reduce the work to that of searching for $P^r_{1...i-1}$ starting from node $RNode(Cid_{i,j})$. This final partial search has to be done using the $child_r(node, a)$ operation repeatedly (once per node arrived at).

On average, we expect that strings present in $RevTrie$ whose reverse is not in $LZTrie$ be of length close to ℓ. The probability of remaining in $RevTrie$ after this length decreases exponentially with m, so we expect the range of lengths of these strings to be $O(1)$. This means that we expect

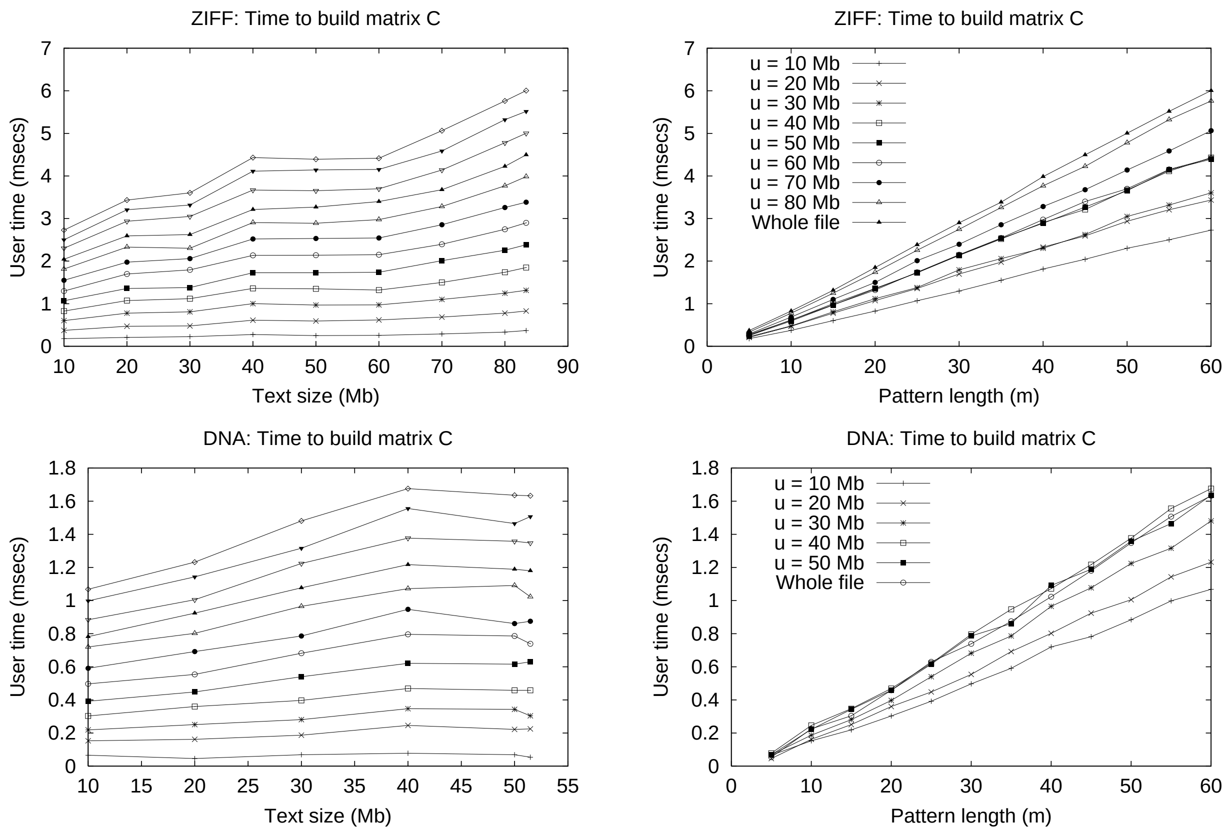


Figure 24: Time to build matrix $C_{i,j}$. On the left, the lines represent, from lower to upper, $m = 5$, $m = 10$, and so on until $m = 60$.

only $O(m)$ cells computed in order to fill vector B_j.

Figure 25 shows that, to build vector B_j, we are working over $O(m)$ cells. Indeed, we work over a 10% of the cells filled to build matrix $C_{i,j}$.

Most of these cells, however, are built via a new mapping to $LZTrie$, and hence several $parent()$ operations are necessary, one sequence per trial until we find the correct child in $RevTrie$. The number of $parent()$ operations to perform per each trial is proportional to the length of the string searched for, $O(\log u)$. On the other hand, we perform a sequence of $parent()$ operations and then may discover that the $RevTrie$ edge was not the good one, and must keep trying until finding the right child. However, at depth $O(\log u)$, we expect $O(1)$ children, so we expect to find the correct child (or not) in $O(1)$ trials. Overall, we expect $O(m \log u)$ work to fill vector B_j.

Figure 26 shows the number of $parent()$ operations performed per $child()$ trial. The case $m = 5$ is excluded from the figures because no cells of B_j were worked on in that case, as all could be predicted using $C_{i,j}$ (case $m < \log u$). The logarithmic pattern is clear.

We also confirmed experimentally the hypothesis that we usually find our child in the first trial. This occurs almost always on DNA and after 1.3–1.5 trials on ZIFF. The figures are higher for $m = 10$ on DNA (1.8) and for $m = 5$ on ZIFF (3.8). However, since the overall cost is negligible for

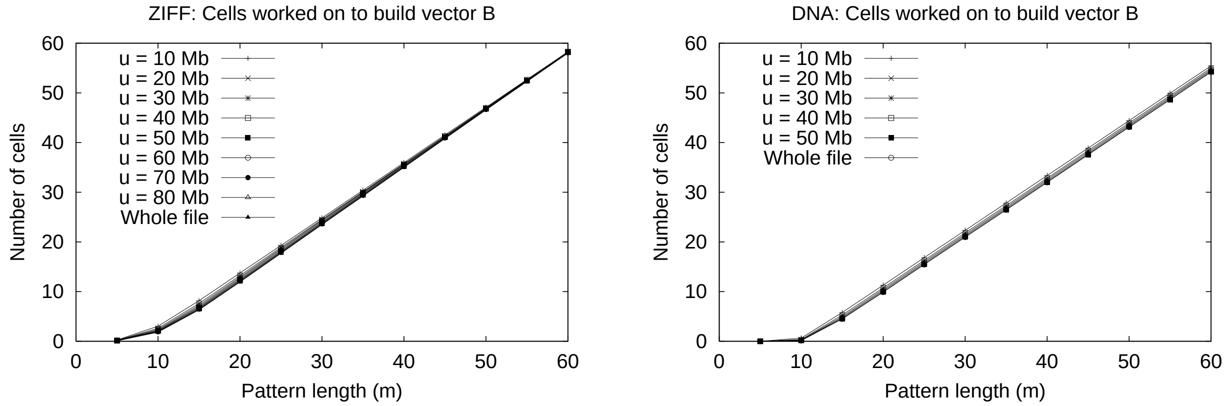


Figure 25: Cells worked on to build vector B_j.

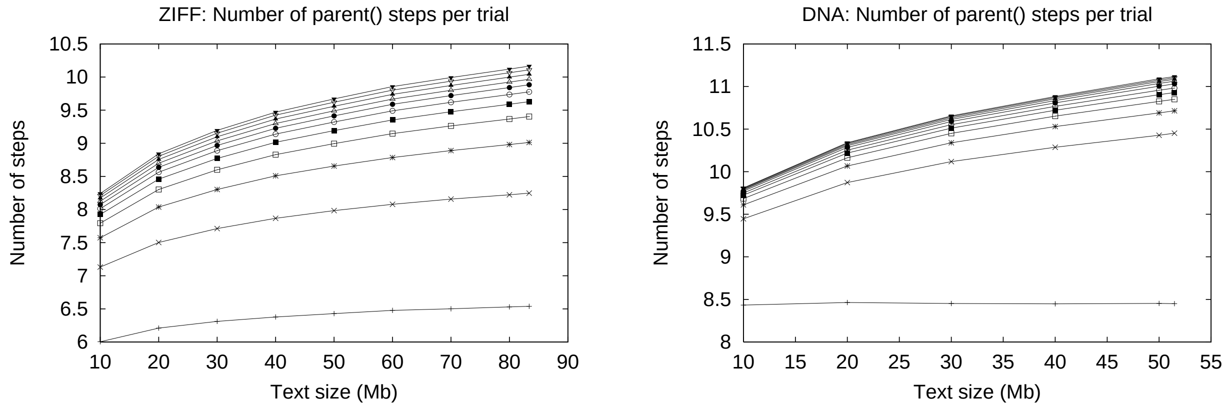


Figure 26: Number of *parent*() operations per *child*() trial. The lines represent, from lower to upper, $m = 10$, $m = 15$, and so on until $m = 60$.

small m values, we can safely disregard these cases.

Figure 27 shows the time to fill vector B_j. As it can be seen, it is much smaller than the time to fill $C_{i,j}$ (at least 3 times smaller on DNA and 10 on ZIFF), despite that in principle filling B_j is much more complex. This is an achievement of our techniques to reduce the amount of computation as much as possible. It can also be seen that the times are essentially linear in m and have a small dependence on u.

Using least squares with the model $t = a + bm + cm \ln u$ (as the logarithmic pattern of Figure 26 is of the form $x + y \ln u$) we obtain

$$\begin{aligned} \text{ZIFF} &= -62.442 + 4.920m + 0.299m \ln u \quad \mu secs \\ \text{DNA} &= -100.433 + 6.752m + 0.271m \ln u \quad \mu secs \end{aligned}$$

with a percentual below 10% in both cases. In this estimation we excluded the values for $m = 5$ for obvious reasons.

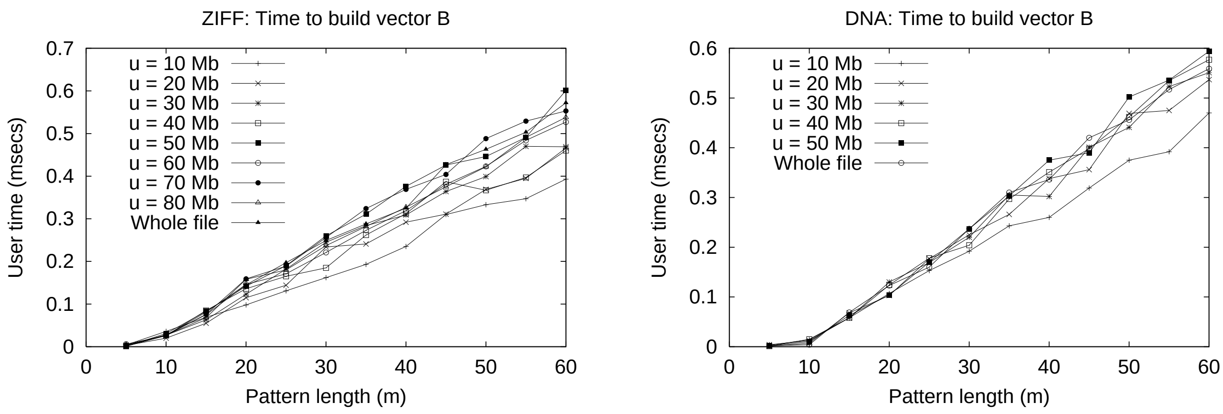


Figure 27: Time to build vector B_j.

6.8.3 Reporting Occurrences of Type 1

We simply consider B_m and, if it exists, $leftrank(B_m)$ to $rightrank(B_m)$. For each block identifier k in this range, we obtain with $Node(k)$ the corresponding $LZTrie$ node. Then the full subtree of $LZTrie$ is traversed, reporting all its nodes (if we just want to count the occurrences, $subtreesize$ is enough).

Figure 28 shows the number of occurrences of type 1. As can be seen, these are significant only for short patterns. The probability of an occurrence being of type 1 is that of a block beginning not falling at positions 2 to m, i.e. $(\ell - m + 1)/\ell$. This is confirmed in Figure 36, where the ratios for $m = 5$ are 0.66 on DNA (predicted 0.66) and 0.58 on ZIFF (predicted 0.60).

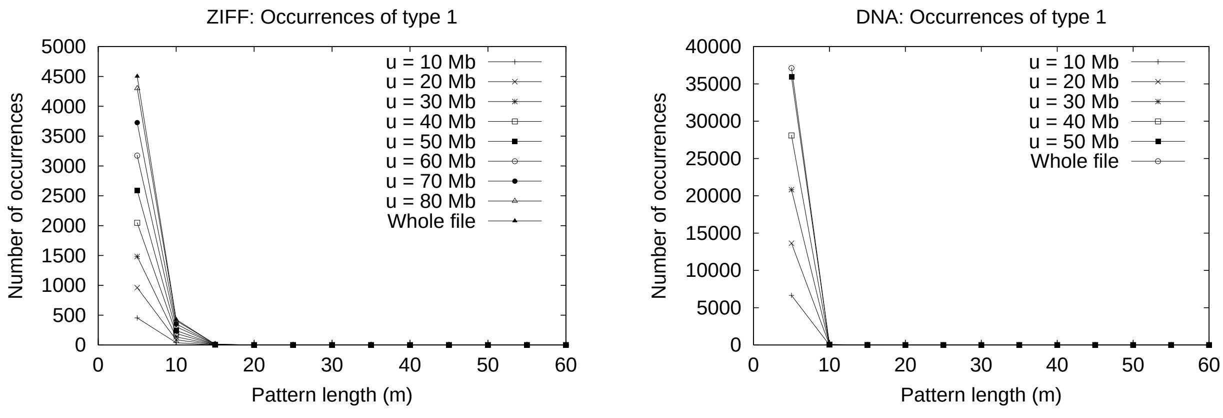


Figure 28: Number of occurrences of type 1.

Figure 29 shows the times. As it can be seen, these are significant only for very frequent patterns ($m = 5$), as the other ones almost have no occurrences of this type in the text. It is also clear that the time to report the occurrences is almost 15 times that of just counting them (in part because we can count whole subtries in one shot). We count about 15,000 occurrences per msec, and report about 1,200 per msec.

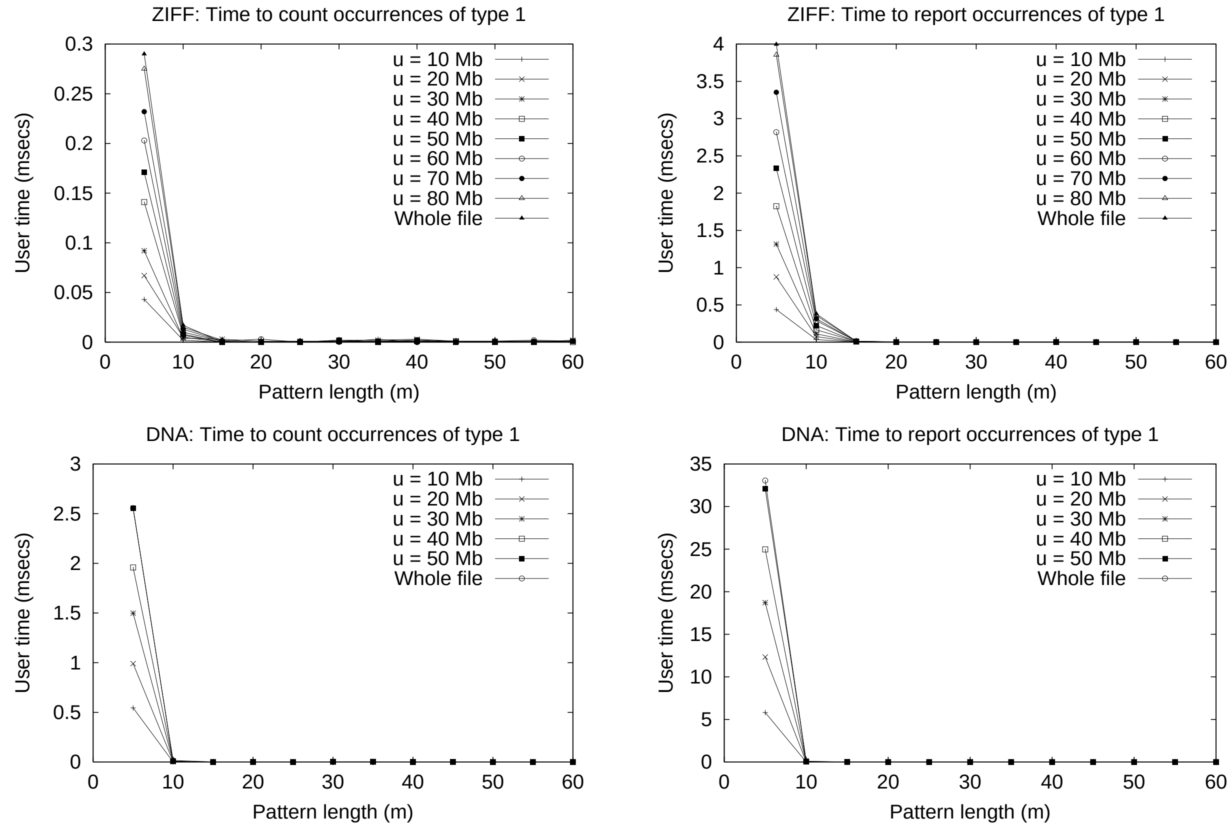


Figure 29: Time to count (left) and to report (right) occurrences of type 1.

Deducting the time of counting from that of reporting, we can write

$$\begin{aligned} count_1 &= 0.067\ (\ell - m + 1)/\ell\ R \quad \mu secs \\ report_1 &= 0.83\ (\ell - m + 1)/\ell\ R \quad \mu secs \end{aligned}$$

6.8.4 Reporting Occurrences of Type 2

As explained when we introduced $RNode$, we have to consider every possible splitting position i (whose nodes are $C_{i+1,m}$ and B_i) and obtain $leftrank$ and $rightrank$ of both nodes. Then we choose to iterate over the smaller range, and for each block identifier, we map the adjacent block number to the other tree using $Node$ or $RNode$. Each time the mapped node descends from $C_{i+1,m}$ (in $LZTrie$) or B_i (in $RevTrie$), depending on our choice, the block is reported.

Figure 30 (left) shows the number of occurrences of type 2. On the right we show the amount of verification work, that is, number of candidate nodes tested. On ZIFF, 1 out of 3.5 becomes a real occurrence, while on DNA only 1 out of 9.5.

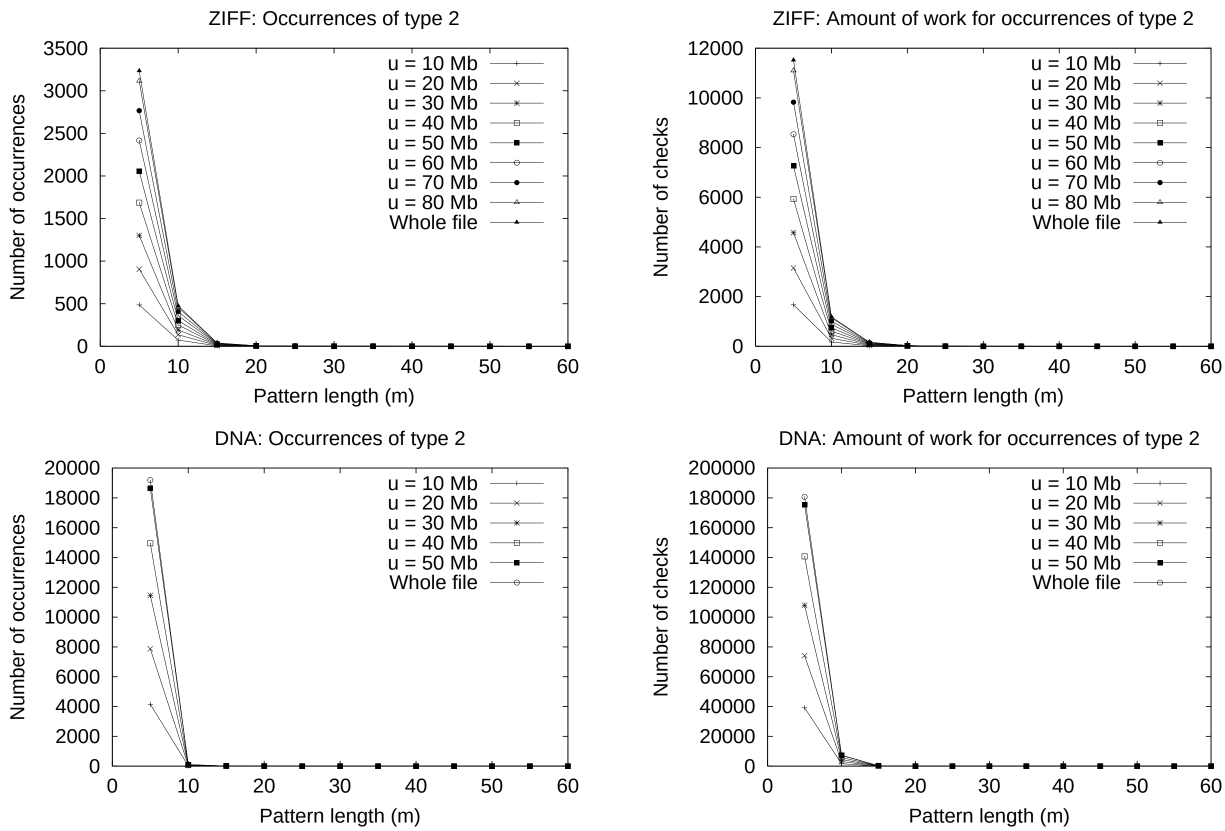


Figure 30: On the left, number of occurrences of type 2. On the right, amount of verification.

Figure 31 shows the times. Again, they are significant only for frequent patterns ($m \leq 15$), as the other ones almost have no occurrences of this type in the text. This time, however, longer patterns have significant times.

The real time for this step depends both on the number of candidates and of real occurrences. We test about 3,500 candidates per msec, considering counting time. Reporting these real occurrences is done at a rate of 1,400 per msec. There is an extra work to test all the splitting positions, but this turns out to be absolutely negligible. Occurrences are reported a bit faster than those of type 1 because in this case we have direct access to the nodes, without the need of traversing the trie. However, the difference is not much significant.

The main issue is to predict how much verification work will we perform. The most important length we have to care about is $m/2$, as the number of verifications triggered by the longer ones is much smaller. We can expect to find $R = u/\sigma^m$ occurrences, while the number of candidates is on average $u/\sigma^{m/2}$. From these, however, only 1 out of ℓ is placed at a text position such that there is a block boundary at position $m/2$. Hence we expect only $u/(\ell\sigma^{m/2})$ candidates, which is $\sqrt{uR}/\ell$. This estimation turns out to work more or less well for DNA, where it predicts 80% of the real amount of verifications, but very bad on ZIFF, where it predicts 7 times the real amount of verifications.

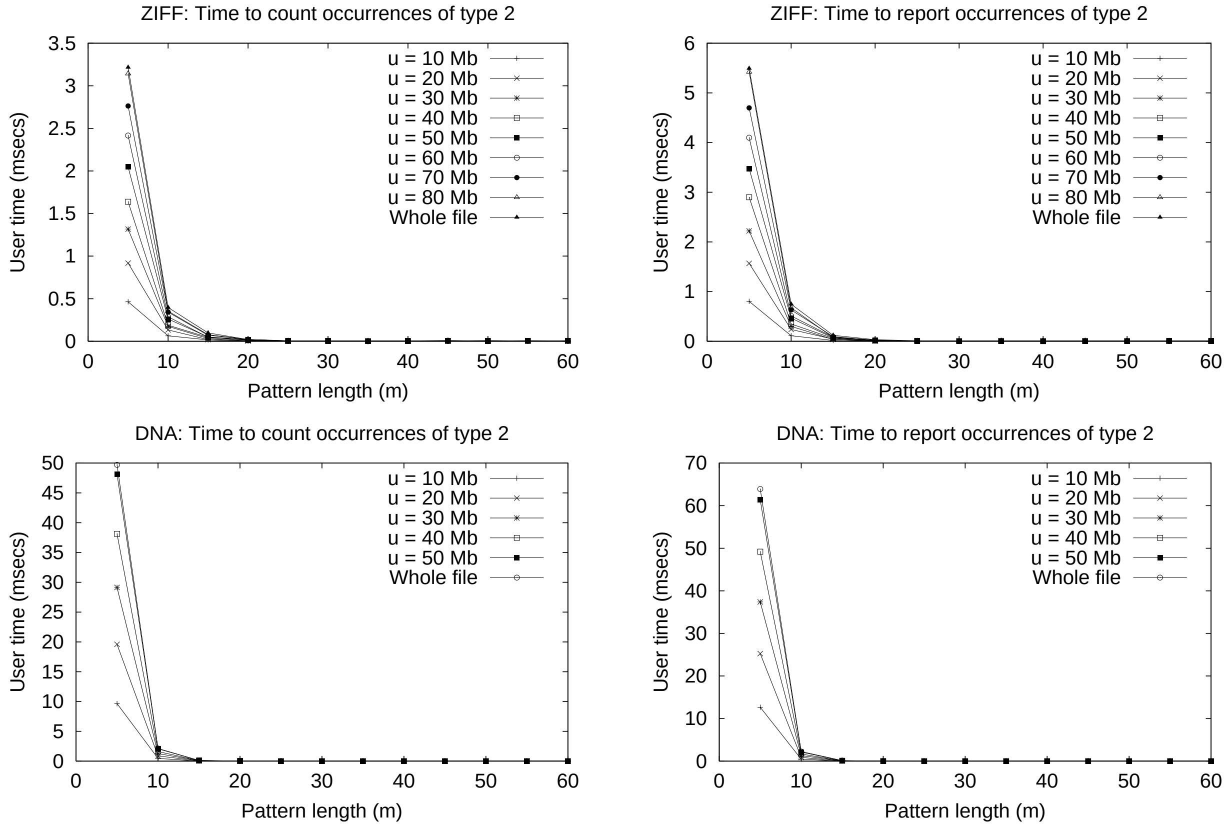


Figure 31: Time to count (left) and to report (right) occurrences of type 2.

This has mostly to do with the fact that the text is not random. For example, the previous figures show that if we know the first or last 3 letters of a text passage, we could guess the next 2 with probability 1/3.5 on ZIFF and 1/9.5 on DNA. This shows that ZIFF has indeed smaller entropy than DNA when we consider longer strings. Table 1 shows the inverse probability of two characters matching as we consider higher order models, as far as we could go with our computer. While DNA stabilizes around 3, ZIFF reaches 1.94 without signs of stabilization.

Let us call σ_k the inverse probability of matching if we consider k previous characters (hence $\sigma = \sigma_0$). We expect that 1 out of $\sigma_3\sigma_4$ candidate occurrences become real occurrences for $m = 5$. That is, the 4th character has to be equal given the first 3, and then the 5th has to match given the first 4. This gives us 1/4.48 on ZIFF and 1/8.82 on DNA. This is a much better estimation, especially because for memory limitations we could not compute exactly the value σ_4 on ZIFF. We expect therefore, at counting time, a verification effort over $R_2W = R_2\sigma_{\lceil m/2\rceil}\ldots\sigma_{m-1}$ candidates, where R_2 is the number of occurrences of type 2. On a random text $\sigma_k = \sigma$ and hence $RW = R\sigma^{m/2} = u/(\ell\sigma^{m/2}) = \sqrt{uR}/\ell$ as explained. We can thus predict

$$
\begin{aligned}
count_2 &= 0.29\ (m-1)/\ell\ RW \quad \mu secs \\
report_2 &= 0.71\ (m-1)/\ell\ R \quad \mu secs
\end{aligned}
$$

Order	ZIFF	DNA
0	20.00	4.82
1	7.31	3.07
2	3.53	3.05
3	2.31	2.98
4	< 1.94	2.96

Table 1: Inverse of the probability of two characters matching, given contexts of different orders.

6.8.5 Reporting Occurrences of Type 3

Instead of the arrays A proposed in the theoretical part, we opt for a closed hash table (load factor at most 1/2) where all the triples $(i, j, Cid_{i,j})$ are stored with key $(i, Cid_{i,j})$ (of course only when Cid is not null). Then we try to extend each match $C_{i,j}$ by looking for $(j+1, j', Cid_{i,j}+1)$ in the hash table, marking entries (i, j) already used by a sequence that starts before, until we cannot extend the current entry. At this point, if the pattern spans 3 blocks or more, the sequence of involved blocks is $k \ldots k'$, and the pattern area is $i \ldots j'$, then we check that $ancestor(C_{j'+1,m}, Node(k'+1))$ holds in $LZTrie$ and that $ancestor(B_{i-1}, RNode(k-1))$ holds in $RevTrie$. If all these tests pass, we report block $k-1$.

As there are $O(m\ell)$ filled cells in $C_{i,j}$, we expect to work $O(m \log u)$ overall in this step. The reason is that the probability of being able of extending a given cell is rather low, hence we can on average extend a cell $O(1)$ times (usually zero).

Figure 32 shows the amount of probes to extend maximal occurrences (left) and number of maximal occurrences found and checked (right). As expected, both are very similar, and they increase linearly with m and slightly with u.

Figure 33 shows the times. They are rather negligible overall, although they increase linearly with m and slightly with u. The number of occurrences found is very low, so the time is basically that to extend occurrences and check maximal ones. Our estimation is:

$$\begin{aligned} \text{ZIFF} &= -15.000 + 0.429m + 0.109m \ln u \quad \mu secs \\ \text{DNA} &= -17.342 + 0.729m + 0.095m \ln u \quad \mu secs \end{aligned}$$

with percentual errors below 20%–25%.

It is somewhat interesting to see the number of occurrences of type 3, shown in Figure 34. It first increases (because short occurrences cannot span 3 blocks) and then decreases as occurrences are less probable. At the end it converges to 1 because patterns are taken from the text. In all cases they are very few.

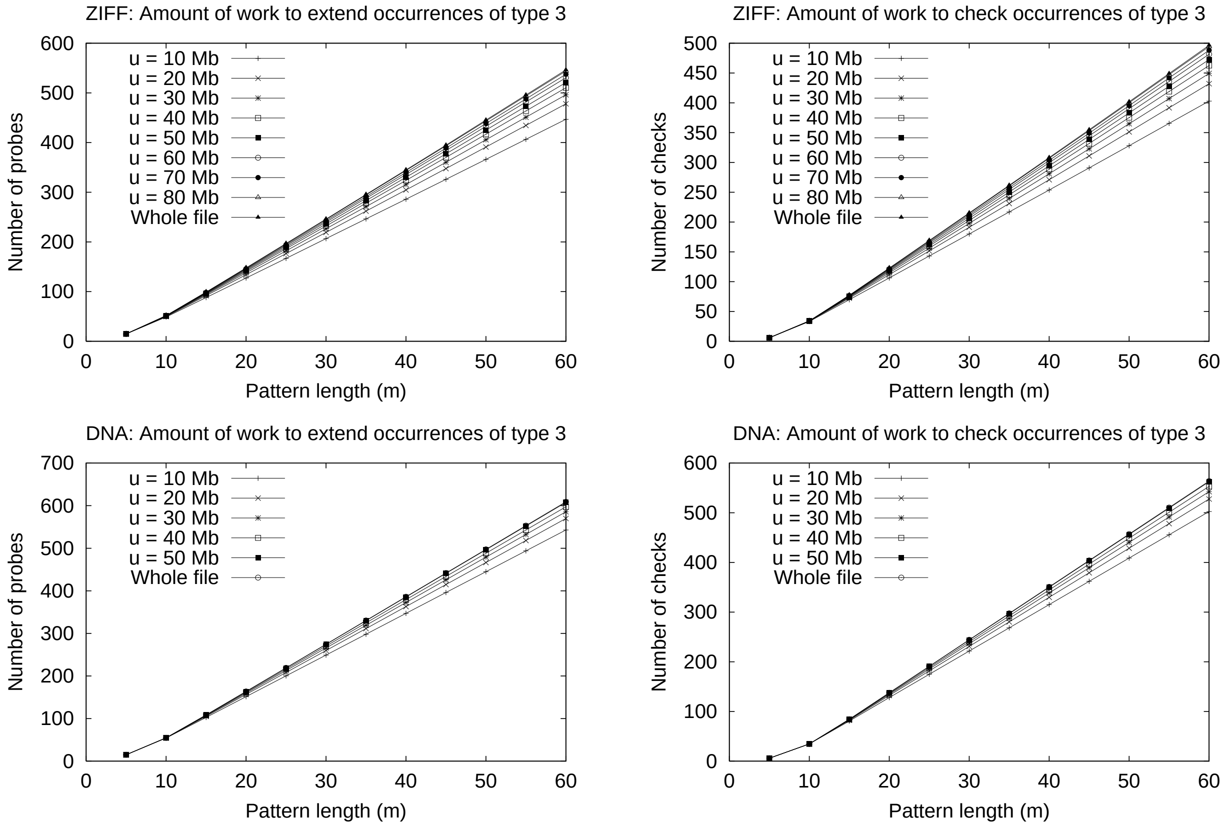


Figure 32: On the left, number of probes to extend maximal occurrences of type 3. On the right, number of checks of maximal occurrences.

6.8.6 Reporting Levels

Reporting is done at three levels. The lowest level is counting: We just tell how many times P occurs in T. The next level (which we have called just "reporting") gives all the text positions of P in T. These are given in the form (block number, offset). Converting them to usual text positions would require another array that maps block identifiers to text positions. This can be done, but we consider that it is not necessary. The representation we use can be used to compare two positions so as to determine which is smaller, and to obtain the surrounding text given a position in that format. This should be enough for most applications. Finally, the highest reporting level prints the text around each occurrence found. Currently the context is limited by newlines, but this is not hard to change. This text is obtained backwards by moving from the block of interest towards the root of $LZTrie$ and printing the letters found at the edges. More and more blocks to the left or to the right are easily obtained using $Node(k-1)$ or $Node(k+1)$.

Figure 35 shows the overall query times under the different reporting levels. Note that we use a logarithmic scale on y. The nonmonotonic behavior comes from the fact that, for short patterns, reporting time largely dominates, while for long patterns the most relevant time is that of searching for the pattern substrings.

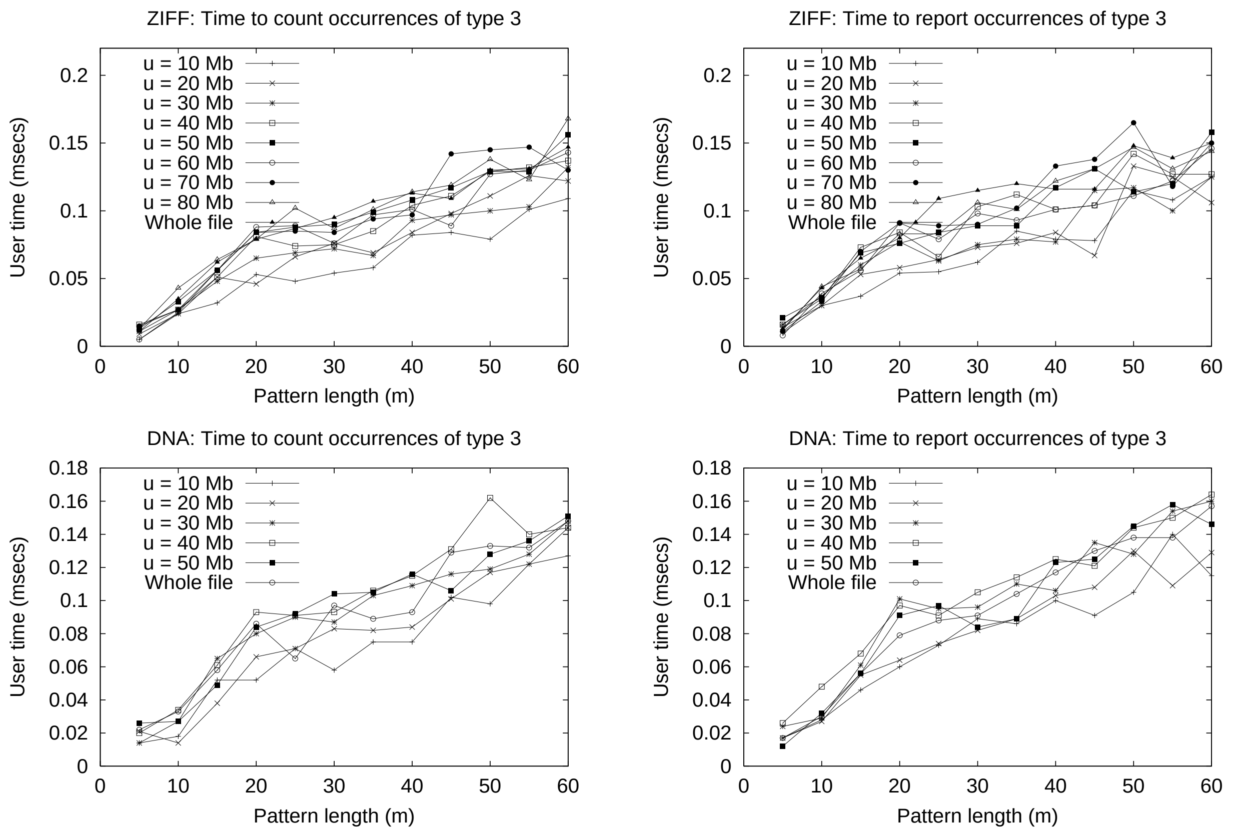


Figure 33: Time to count (left) and to report (right) occurrences of type 3.

Figure 36 shows the overall number of occurrences. This shows that we report 600–800 occurrences per msec, and output about 14 matching lines per msec.

7 Comparison Against Others

We have compared our prototype against two of the most prominent alternative proposals. We have used the same experimental setup of the previous section, taking the whole ZIFF and DNA texts. We have considered construction time and space, but our highest interest is in query times, both for counting and for reporting.

An important fact is that the different indexes take different space. Although our index does not permit important space-time tradeoffs[7], the others do. Hence, we tune the other indexes so as to make them take the same space of our index.

All these indexes need to operate the data structure in main memory, as their access pattern to the structures is random. Hence, it should be made clear what we mean by "the space of the index", as we can usually store them taking less space than the one needed to operate in main

[7] Some could be achieved by enlarging hash tables, for example, but the result is not much significant.

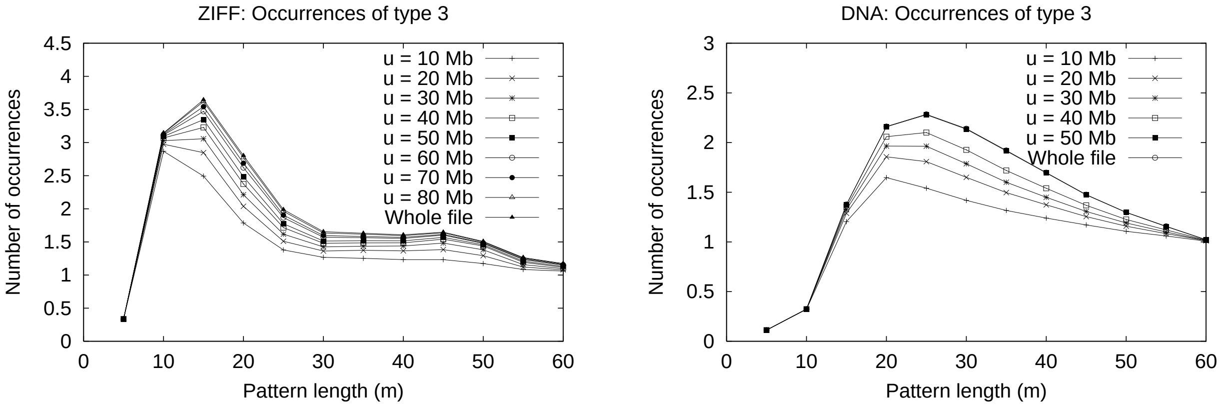


Figure 34: Number of occurrences of type 3.

memory. In one extreme, we could just map all our main memory data structures to disk and use the same disk space as the main memory space, and "booting" the index would consist of directly loading all the disk space into main memory. On the other extreme, we could simply store the plain text compressed in the best way, totally independent of our index, and "booting" would consist of actually constructing the index. As the server probably will load the index once and then answer many queries, both choices may be acceptable. In particular, depending on the index construction time versus disk speed, one or the other extreme may be the best choice. This shows that speaking of the space needed by the index on disk may have little sense, as all could just store the compressed text and build the index on the fly. Rather, we will be interested in how much main memory space does the index need in order to operate properly.

We will first explain the implementations of the alternative indexes chosen, and then show the results of the comparison.

7.1 Ferragina and Manzini's FM-index

This index, proposed in [6, 7], consists of (i) the permuted text, whose characters are reordered according to the Burrows-Wheeler transform and then compressed using run-length and move-to-front, (ii) a two-level directory C that permits knowing $count(c, i)$, the number of times character c appers before position ir in the permuted text, and (iii) a sampling of pointers S that tells, for 1 out of D permuted text positions, which is their position in the original text.

We could not obtain the sources of the implementation of this index from the authors. There is an executable at their Web page, `http://butirro.di.unipi.it/ ferrax/fmindex/index.html`, but the interface does not permit running massive and trustable tests, as it can search for one pattern per run. Hence, we implemented the index ourselves. We explain now the decisions taken. These follow rather closely the descriptions in [7]. We did our best to implement this index as efficiently as possible. Later we will give some control values to show that our implementation is competitive against the executables given by the authors.

Directory C works as follows. The permuted text is divided into blocks of 2^8 characters and superblocks of 2^{16} characters. For each superblock and character c we store a cumulative count

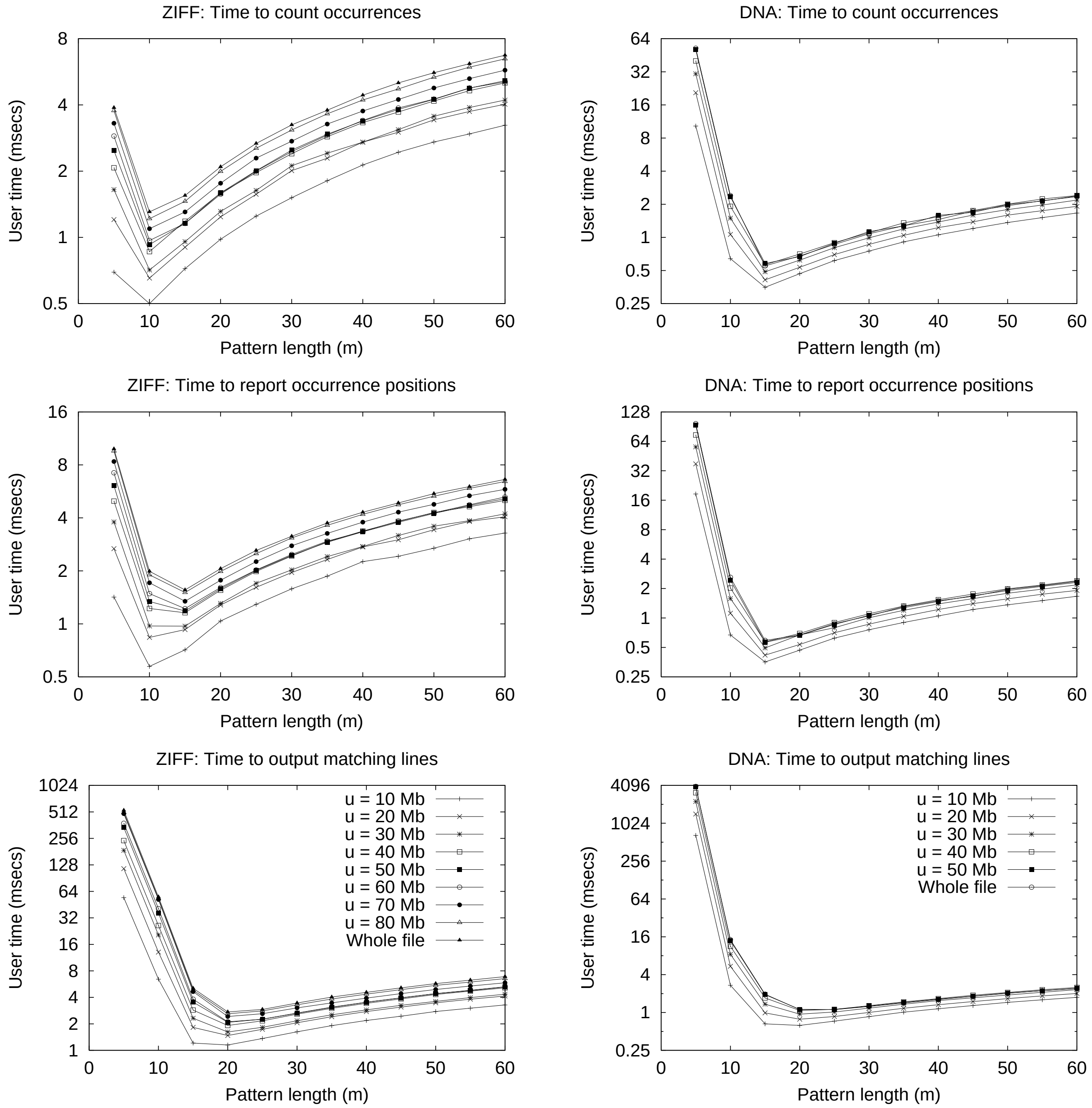
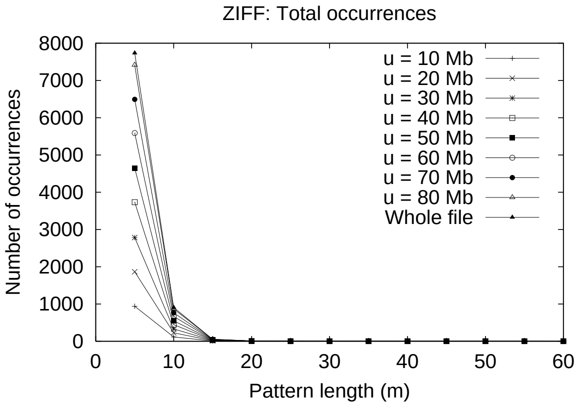


Figure 35: Overall query times when counting occurrences (top), reporting positions (middle), and to output matching lines (bottom). The legends are on the lowest figures of each column.



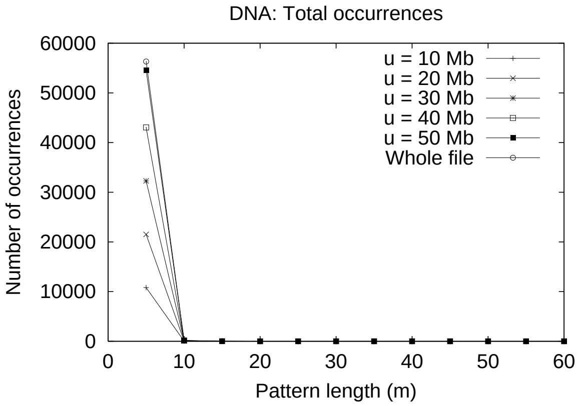


Figure 36: Overall number of occurrences.

of the occurrences of c up to the beginning of the superblock. Inside each superblock, for each character c, we store a byte sequence (of length at most 2^8) of how many times does c appear inside each block of that superblock. Runs of zeros are frequent and these are run-length compressed. Hence, in order to compute $count(c, i)$ we have to (a) get the superblock counter value; (b) run through the byte sequence of c inside the superblock, adding per-block occurrences until reaching the relevant block (runs of zeros are traversed fast); and (c) traverse the permuted text inside the block to add up the final intra-block value. This structure permits computing $count(c, i)$ by doing simple operations over at most $2^8 + 2^8$ bytes (considering byte sequences and permuted text). Its extra space requirement is rather modest, as seen soon.

The sampling of pointers S is used only when reporting occurrence positions. It is implemented as a plain array of integers, which is appropriate for the text sizes we are considering (we would have needed 26–27 bits out of 32 anyway, and access is much faster this way). The value of D directly affects reporting times, because we need to perform, on average, $D/2$ $count(i, c)$ operations in order to discover the text position of each occurrence.

The idea is that, starting at some permuted text position, we move backwards position-wise in the *original* text positions, which leads us to another (basically random) permuted text position. We keep doing that until we find a position that is sampled in S, and then we know where we are in the original text and where we were when we started the process. S regularly samples the original text, so we need, in addition to the array of pointers, a hash table that answers whether a given permuted text position is sampled or not. This was implemented as a closed hash table with load factor 0.5. The idea suggested in [7], of marking characters and sample their positions, was discarded because it does not permit obtaning the desired extra space on DNA.

There are two possible space-time tradeoffs for this index. A first one refers to the amount of information on blocks and superblocks, as smaller ones reduce the cost of $count(c, i)$ operations. A second one is the value D, which is only meaningful for reporting queries.

For counting queries, we should drop the array of pointers S and use more space for C. However, this index turned out to be so fast for these queries that there was not a point in optimizing it for this case. As we see shortly, our index is completely out of competition for these queries.

For reporting queries, spending more space on S reduces the search time faster than spending it on C. Hence we used C as explained, using fixed space, and reduced D as much as possible until using the permitted amount of space. In order to give the FM-index the same main memory of our index, we let it use an additional 8.1% of the text space to store S on ZIFF and 3.4% on DNA. This means a sampling of 1 out of 50 entries for ZIFF and 1 out of 120 for DNA. With these parameters, directory C took 4.82% of the total index space on ZIFF and 1.89% on DNA, and the sampling of S took 27.59% of the index space on ZIFF and 14.32% on DNA. The rest is used for the permuted text.

The percentages given to D are so low because we decided to store the permuted text in uncompressed form. The other choice would have been keeping it compressed and using the extra space for S. The compressed texts take 23.24% of the original text on ZIFF and 25.27% on DNA (we use Huffman to compress the move-to-front values, as this gave better results than δ-coding [4]). This would have permitted us to use, for S, 2.87 times more space on ZIFF and 5.40 on DNA. This means that D would decrease 2.87 or 5.40 times. Multiplying this by the number of characters processed per block, we have that we would have processed 12.35 times fewer bytes on ZIFF and 21.37 on DNA.

The drawback, of course, is that those bytes we have to traverse now are much more expensive to process than uncompressed text characters: They are bit-codings of move-to-front values. Instead of just one comparison and an optional increment over registers, we have to (*i*) extract the bits, (*ii*) obtain their numerical value from the Huffman tree, (*iii*) search the move-to-front linked list (usually a few positions, say 3 or 4), (*iv*) move the element found to front, and (*v*) do the comparison and optional increment as well. Decoding move-to-front is the slowest part of this picture, which can perfectly exceed the 12- or 20-fold gain. Indeed, in our experiments processing each compressed Mb took 387 microseconds on ZIFF and 338 on DNA. Comparing and adding took 4.12 microseconds on ZIFF and 2.06 on DNA. This shows that decompression poses a 100- to 150-fold slowdown, much larger than the 12- to 20-fold speedup.

Hence, we decided to keep the permuted text in uncompressed form. Of course this is not an option if one wants the FM-index to take less space than the text, but this is the best choice in order to compete against our index. It should also be clear that the decision strongly depends on the type of compression used, for the compression ratio and decompression speed. Other schemes would yield a different result. A study on this is interesting as well, but out of the scope of this paper. A different choice, however, is explored in the next section.

Building the index implies a suffix array construction. We have used a simple quicksort, although there are faster methods. We are not focusing much on index construction times anyway.

7.2 Sadakane's CSArray

We obtained from K. Sadakane his implementation of the Compressed Suffix Array index proposed in [25]. This implements a suffix array search over a compressed representation of it. The main data structures are Ψ, which permits moving from the suffix array position that points to i to that pointing to $i+1$, and a sampling of the suffix array. The array Ψ can be stored with a technique similar to that used by the FM-index to store the text, as values $\Psi(i) - \Psi(i-1)$ tend to be small. It also needs some directory structures to compute cumulative frequencies, just as the FM-index. The sampling of the suffix array plays a similar role as well. As can be seen, the ideas are not so radically different after some analysis. This index permits moving "forward" in the text, while the

Index	Construction time		Main memory space	
	ZIFF	DNA	ZIFF	DNA
FM-index	4.990	5.260	5.00	5.00
CSArray	19.28	6.890	11.18	10.20
LZ-index	0.968	0.605	4.95	3.46

Table 2: Index construction requirements. Times are in seconds per Mb and space in number of times the text size.

FM-index moves "backward".

One important difference, however, is that the CSArray does not need move-to-front as an intermediate compression step. This permits much faster decompression. Moreover, this implementation uses just δ-encoding, which is decompressed fast. This permits keeping the text in compressed form and using more space for the suffix array sampling, with a much smaller decompression penalty.

Two parameters permit different space-time tradeoffs. The first, D, is the sampling interval of the suffix array: 1 out of D suffix array values are stored explicitly. The second, L, is the sampling step for Ψ (apart from the cumulative differences, we need explicit values from time to time). A counting query takes $O(m(L+\log u))$ time, while a reporting query takes $O(RDL)$ time. We tested a thorough range of combinations of D and L giving the same space of our index. The best in practice turned out to be $D = 7,\ L = 16$ for ZIFF and $D = 8,\ L = 32$ for DNA.

In the construction, Sadakane uses a faster suffix array construction method, which needs 8 times the text size at construction time. However, compressing the resulting suffix array takes even more space. Probably this has not been optimized for this prototype. We only rewrote small parts of the code in order to improve the way text contexts around occurrences are shown.

7.3 Comparison

Recall that we compare the three indexes such that they take the same amount of main memory to function. Table 2 shows the time and memory requirements to build the different indexes (although the final index space is the same, they need different space to build). As it can be seen, our index (LZ-index) builds much faster than the others (whose construction time involve at least the construction of a suffix array). It also needs less memory to build.

Let us now consider search times. Figure 37 shows the overall query times under the different reporting levels. Note that we use a logarithmic scale on y.

For counting queries, the FM-index is unparalleled, taking around $1.7m$ μsecs. The CSArray, although slower, is still much faster than our LZ-index, taking around $5m$ μsecs. It is clear that we do not have a case for counting queries: the LZ-index took $112m$ μsecs on ZIFF and $38m$ μsecs on DNA, 10–20 times slower than the CSArray and 20–60 times slower than the FM-index.

The FM-index, however, becomes *much* slower to report the positions of the occurrences found, achieving a rate of 10–20 occurrences per msec (recall that our rate is closer to 900–1,400 per msec, depending on the hit ratio on candidates of type 2). The CSArray is faster than the FM-index at this step, reporting 100–160 occurrences per msec, by taking advantage of the denser sampling.

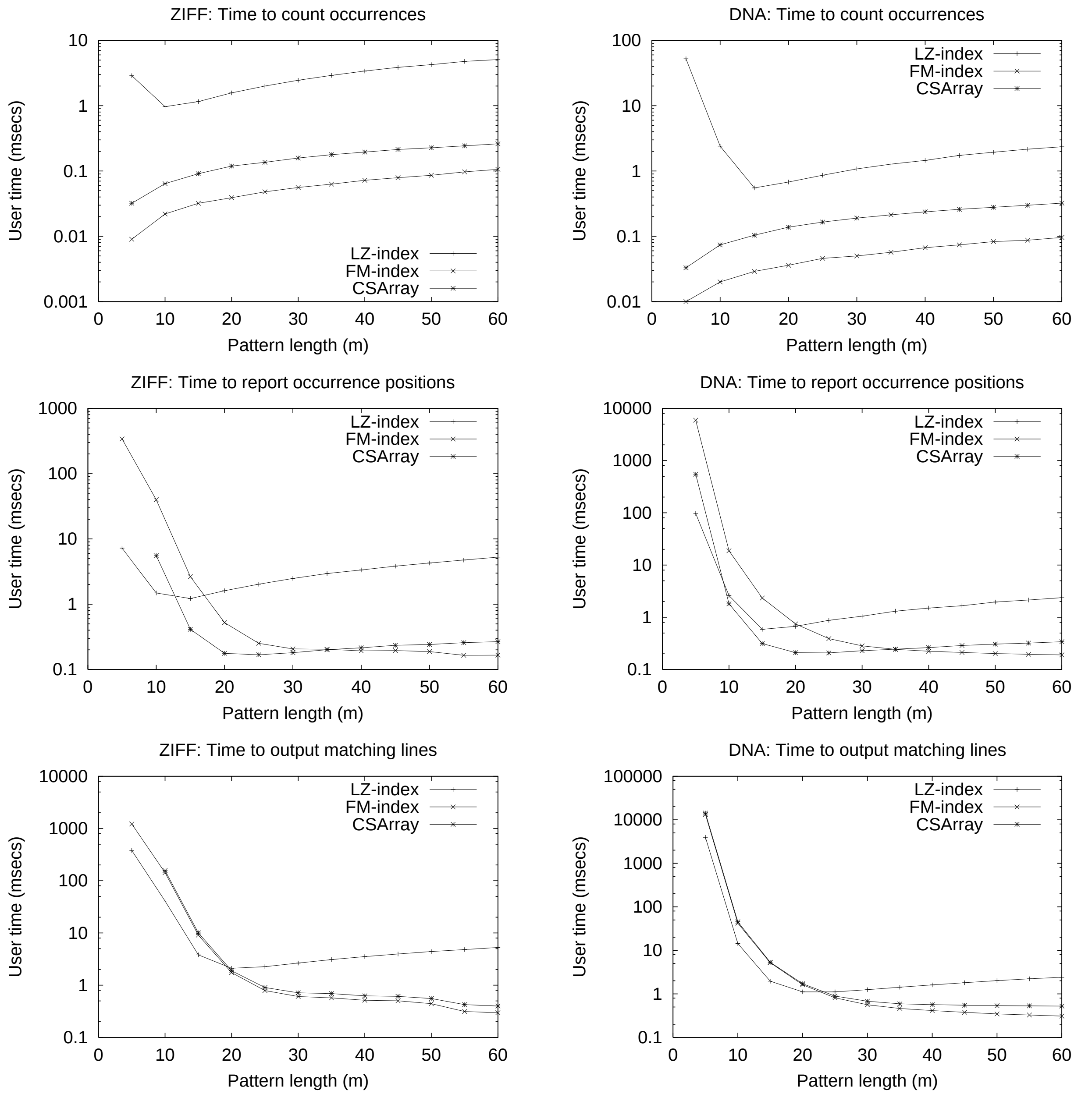


Figure 37: Overall query times when counting occurrences (top), reporting positions (middle), and to output matching lines (bottom). We compare our LZ-index against the most relevant alternatives.

This is, in turn, a consequence of the different compression methods used. In any case, it is clear that finding the actual positions of the occurrences is costly under their schemes, 70–90 times slower for the FM-index and 9 times slower for the CSArray.

The differences favor the LZ-index even more if we ask to reproduce the lines where the occurrences were found. Remind that this is an essential feature, since all these indexes *replace* the text and hence our only way to see the text is asking them to reproduce it. While our LZ-index is able to show around 14 lines per msec, the FM-index and the CSArray can show only 4–6 lines per msec. This time the sampling of the suffix array plays a less important role (and hence our FM-index and CSArray implementations have little differences). The reason is that the sampled suffix array is used just to find the occurrence position, and from then on one letter is output per step (forward or backward move in the original text). The problem is that each such step needs to find the character that corresponds to that suffix array position.

As a conclusion, we have that our index is rather slow to count the number of occurrences, but much faster to show their positions or their text contexts. This is rather intrinsic, because in our index the occurrences of P are scattered all around the index, while these are all together in a suffix array. Giving the occurrence positions and text contexts, however, is rather fast because we did most of the work in the counting phase. We require only a fast tree traversal step per character output. Compressed suffix arrays, on the other hand, rely on a sampled suffix array and they must perform expensive traversals until they determine the actual suffix array values.

We claim that, for most text retrieval needs, knowing just the amount of occurrences is not enough. Although it may be useful at the internal machinery of other more complex tasks, the bottom line is that the user wants to know where the occurrences are and most probably to see their text context (not to speak of retrieving the whole document, not the line, containing the occurrence).

Let us be pessimistic against the LZ-index and assume that one can build an alternative as fast as the FM-index to search for the pattern and as fast as the CSArray to show the occurrences (this scenario is rather realistic). It turns out that, to report the occurrences, the LZ-index would become faster after we report 1,400 occurrences on ZIFF or 300 on DNA. If we would like to see the lines containing the occurrences, these numbers drop to 65 on ZIFF and 13 on DNA. This shows that our index becomes superior as soon as we have to show a few occurrences.

To conclude, we give some data on our tests over the executables of the FM-index provided by the authors. These permit a coarse control over the index space by specifying the frequency of a character whose positions will be sampled. Although we tried the highest possible frequencies, we could not obtain indexes larger than 75.02% of the ZIFF file and 109.81% of the DNA file. The former is half the space we permit, while the latter is rather close to the correct value. The main memory required to build the index is 9 times the text size, and the construction speed is 1.7 to 2.3 sec/Mb. The time to count occurrences is negligible, as expected. Occurrence positions were reported at a rate that varied a lot, but was always between 0.5 and 10 occurrences per msec. When we asked the index to show a text context of length equivalent to an average line (43 characters on ZIFF and 61 on DNA), it showed them at a rate of 10 to 20 per *second*. Even if we assume that the index on ZIFF could double its performance by using twice the space, the figures still show that our implementation of the FM-index is competitive against that of the original authors, when not superior by far[8]. The results did not vary when we tried different memory policies offered by the index (on disk, mmaped, in main memory).

8 Conclusions

We have presented an index for text searching based on the LZ78/LZW compression. At the price of $4n\log_2 n(1+o(1))$ bits, we are able to find the R occurrences of a pattern of length m in a text of n blocks in $O(m^3 \log\sigma + (m+R)\log n)$ time.

We have implemented the index and considered the tradeoffs between theoretically good ideas and practically efficient implementations. We have empirically studied the behavior of the many aspects of our index, and shown that the average search cost of our implementation is of the form $O(\sigma m \log u + \sqrt{u}R)$.

Finally, we have compared our prototype against existing alternatives and have shown that our index is competitive in practice. Although it is much slower to count how many occurrences are there, it is much faster to report their position or their text context. Indeed, we show that if there are more than 1,400 (ZIFF) or 300 (DNA) occurrence positions to report, or more than 65 (ZIFF) or 13 (DNA) text lines to show, the LZ-index becomes superior. In our experiments this happened up to $m \leq 10$ (ZIFF) or $m \leq 5$ (DNA) to report occurrence positions and up to $m \leq 20$ (ZIFF and DNA) to report matching lines. This includes most of the interesting cases on natural language and several ones on genetic sequences.

Altough the slowness for counting queries is intrinsic of our index, we believe that times can be at least improved. One clear slowdown factor is the linear search of nodes when executing $child(i,a)$, as the time to fill matrix $C_{i,j}$ dominates the overall time once we exclude reporting. One choice would be to replace it by a two-level structure, where children are grouped into $\sqrt{\sigma}$ contiguous groups of $\sqrt{\sigma}$ nodes each, hence permitting faster access to the desired child. Another operation whose improvement will benefit the overall search time is that of finding matching parentheses ($findclose()$ and $parent()$).

Other challenges that lie ahead are performing regular expression and approximate searching using this index, working on secondary memory, and trying to compete against compressed inverted indexes designed for natural language text. Building the index in succint space would be an important step in this direction (see, for example, [18]).

The implementation of ours and other indexes has pointed out the large gap between theoretically and practically appealing ideas. We have shown several examples where the best decisions do not match. We have also learned that most of the efficiency in a real implementation of these indexes may have to do with aspects that are usually disregarded at the algorithmic level, such as the exact way the compression is performed. The best way to implement these indexes, as well as a thorough comparison, is an interesting open issue.

Acknowledgements

We thank Kunihiko Sadakane for kindly giving as a prototype of his index [25].

[8]We believe that the authors have optimized their implementation for a space consumption much inferior than that of our comparison. For example, they keep the permuted text in compressed form.

References

[1] M. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th Intl. Symp. String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 31–43, 2002.

[2] P. Agarwal and J. Erickson. Geometric range searching and its relatives. *Contemporary Mathematics*, 23: Advances in Discrete and Computational Geometry:1–56, 1999.

[3] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[4] T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.

[5] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symp. Foundations of Computer Science (FOCS'00)*, pages 390–398, 2000.

[7] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th ACM Symp. on Discrete Algorithms (SODA'01)*, pages 269–278, 2001.

[8] P. Ferragina and G. Manzini. On compressing and indexing data. Technical Report TR-02-01, Dipartamento di Informatica, Univ. of Pisa, 2002.

[9] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symp. Theory of Computing (STOC'00)*, pages 397–406, 2000.

[10] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[11] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS'89)*, pages 549–554, 1989.

[12] J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Ann. Symp. Combinatorial Pattern Matching (CPM'95)*, LNCS 937, pages 191–204, 1995.

[13] J. Kärkkäinen. *Repetition-based text indexes.* PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999. Also available as Report A-1999-4, Series A.

[14] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pages 141–155. Carleton University Press, 1996.

[15] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Ann. Intl. Conference on Computing and Combinatorics (COCOON'96)*, LNCS 1090, 1996.

[16] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[17] S. Kurtz. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Universität Bielefeld, 1998.

[18] T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Ann. Intl. Conference on Computing and Combinatorics (COCOON'02)*, pages 401–410, 2002.

[19] V. Mäkinen. Compact suffix array. In *Proc. 11th Ann. Symp. Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 305–319, 2000.

[20] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

[21] I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, LNCS 1180, pages 37–42, 1996.

[22] I. Munro and V. Raman. Succint representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Symp. Foundations of Computer Science (FOCS'97)*, pages 118–126, 1997.

[23] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, pages 205–222, 2001.

[24] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[25] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Intl. Symp. Algorithms and Computation (ISAAC'00)*, LNCS 1969, pages 410–421, 2000.

[26] K. Sadakane. Succint representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. 13th ACM Symp. on Discrete Algorithms (SODA'02)*, pages 225–232, 2002.

[27] T. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

[28] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, second edition, 1999.

[29] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. on Information Theory*, 24:530–536, 1978.