Indexing Text using 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, which is faster than current alternatives when we take into consideration the time to report the positions or text contexts of the occurrences found.

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 [30, 26] (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 [27] 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 [28] 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. Some decisions are changed in the implementation because of practical considerations. 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). 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. A shorter version of this paper appeared in [24].

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 [31]) 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).

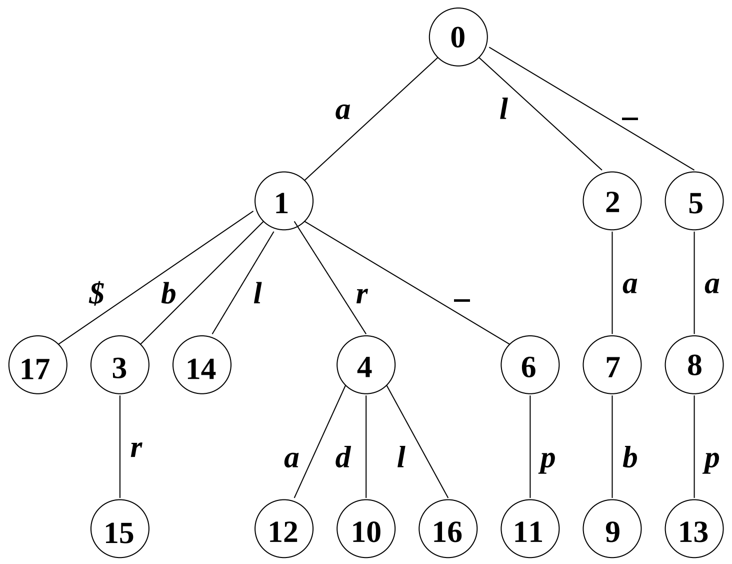
Many variations on LZ78 exist, which deal basically with the best way to code the pairs in the compressed file. A particularly interesting variant is from Welch, called LZW [29]. In this case, the extra letter (second element of the pair) is not coded, but it is taken as the first letter of the next 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, 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

[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.

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.

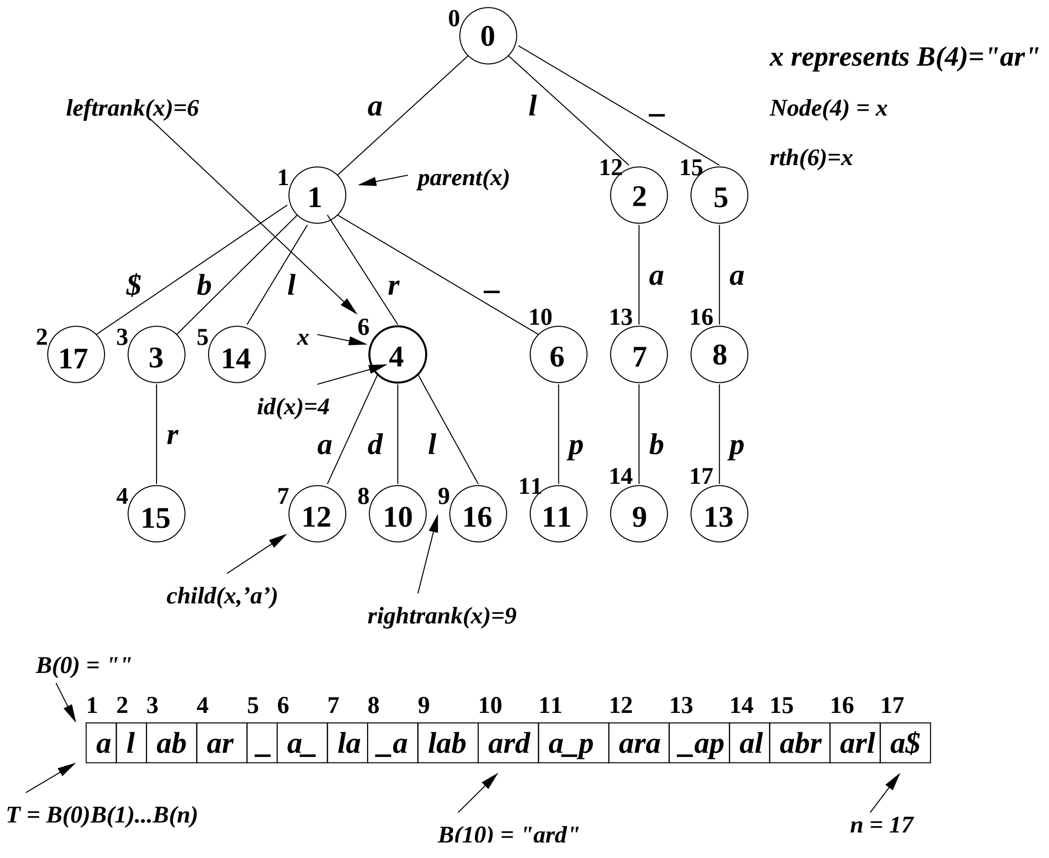


Figure 2: *LZTrie* data structure for our running example. The numbers over the nodes are their rank. We show the values that correspond to node x, which represents block number 4 and is the 6th string in the set.

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.

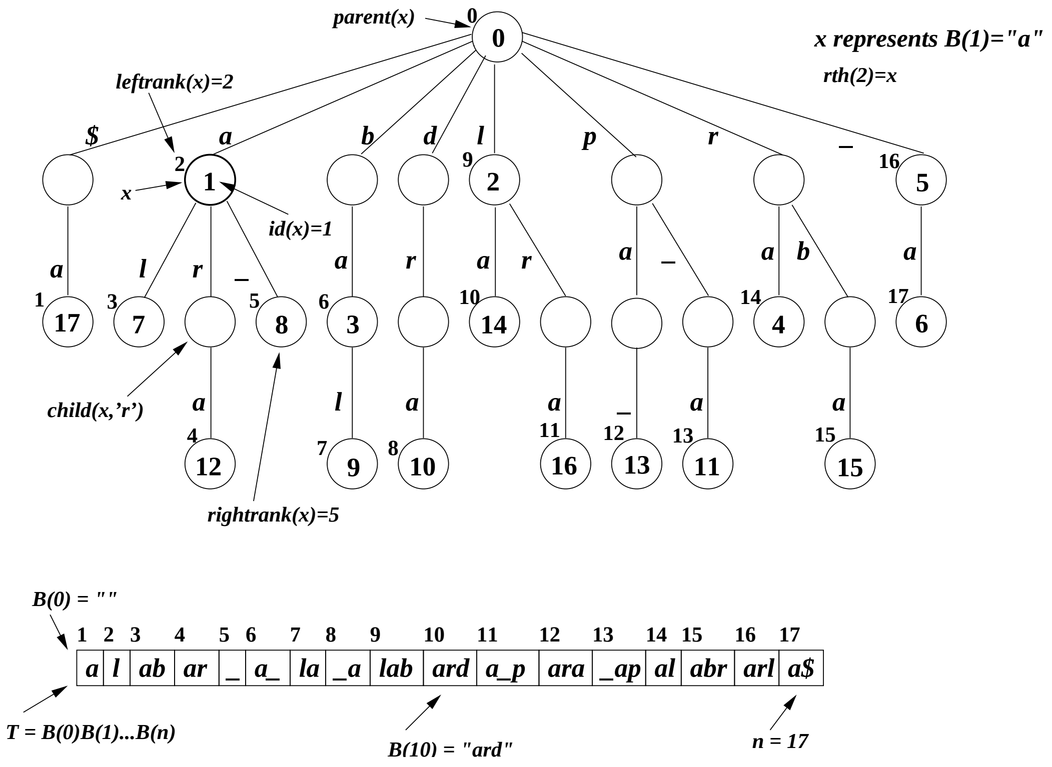


Figure 3: $RevTrie$ data structure for our running example. As we store the reversed strings, the set is not prefix-closed and not every node corresponds to a block identifier. We show the values that correspond to node x, which represents block number 1 and is the 2nd string in the set.

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;

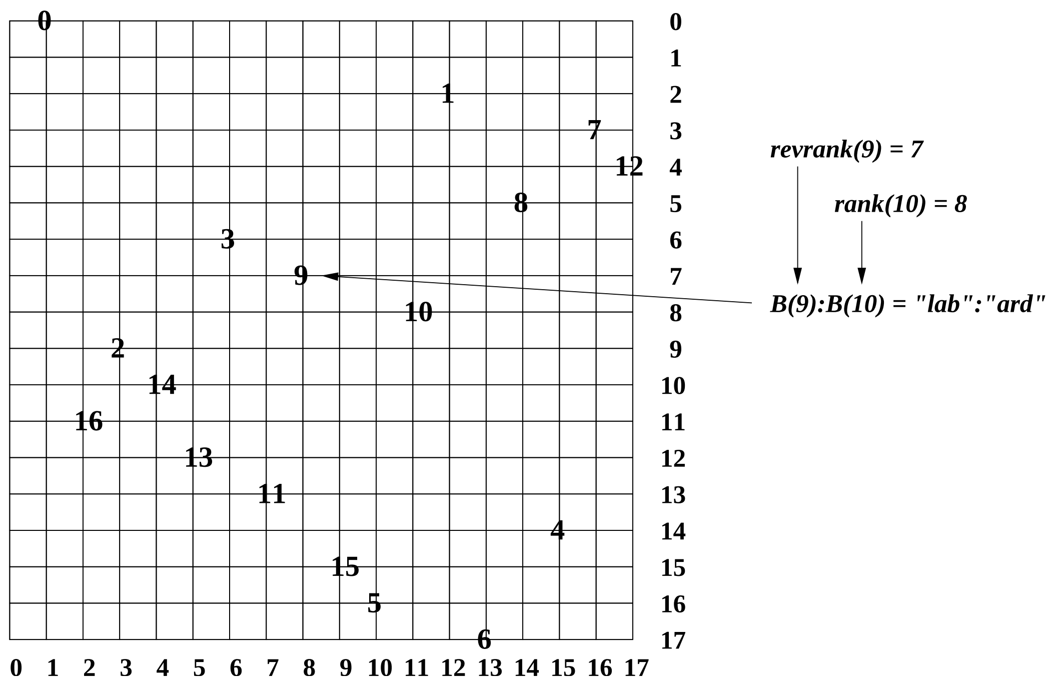


Figure 4: *Range* data structure for our running example. For instance, the pair of consecutive blocks 9:10 have reversed rank and rank, respectively, 7 and 8. Hence block number 9 is stored at row 7 and column 8 of the data structure.

(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

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

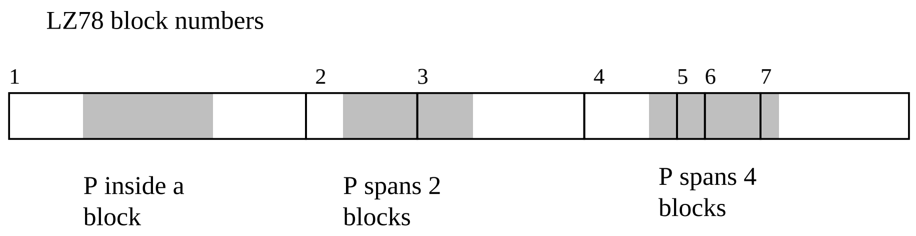


Figure 5: Different situations in which P can match inside T.

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 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 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 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.

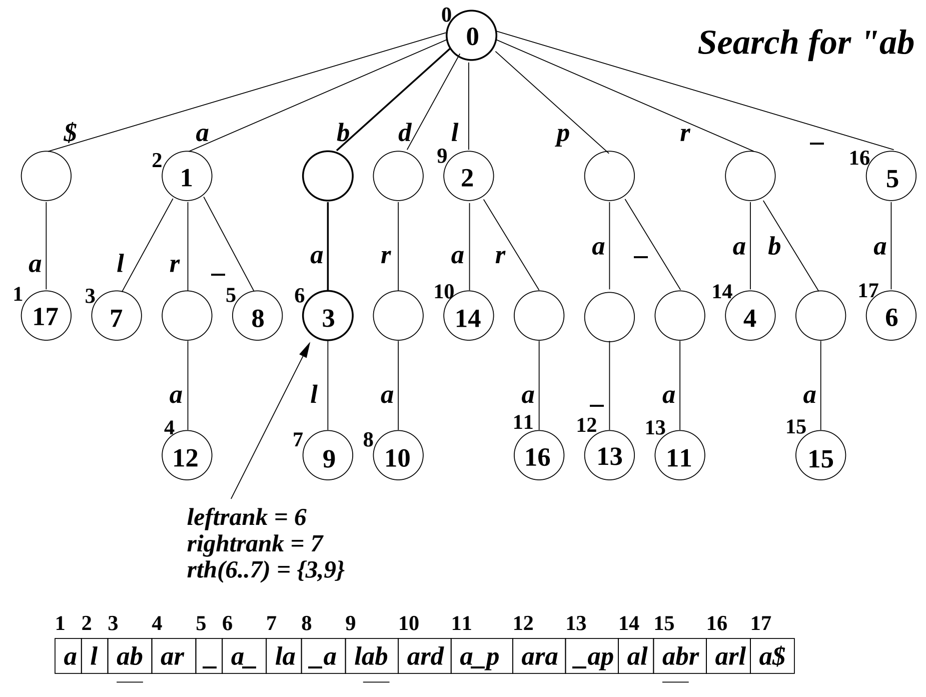


Figure 6: Reporting occurrences of type 1 of $P = ab$ in our running example, first part.

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.

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 $P_{i \ldots j}$ 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 $P_{i \ldots j}$. We store the block numbers in m arrays A_i, where A_i stores the blocks corresponding to $P_{i \ldots j}$ for all

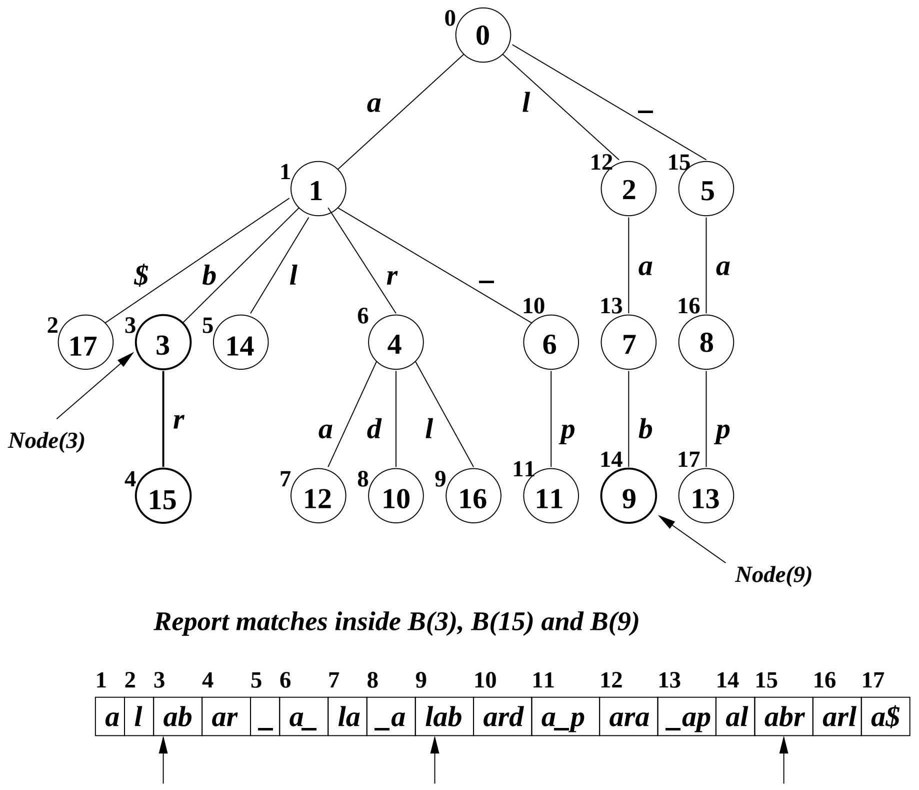


Figure 7: Reporting occurrences of type 1 of $P = ab$ in our running example, second part.

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 $1 \leq i \leq j \leq m$, 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 $j' \geq j$. At the end of every i-turn, we sort A_i by block number. Mark every $C_{i,j}$ as *unused*.

2. For every $1 \leq i \leq j < m$, 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).

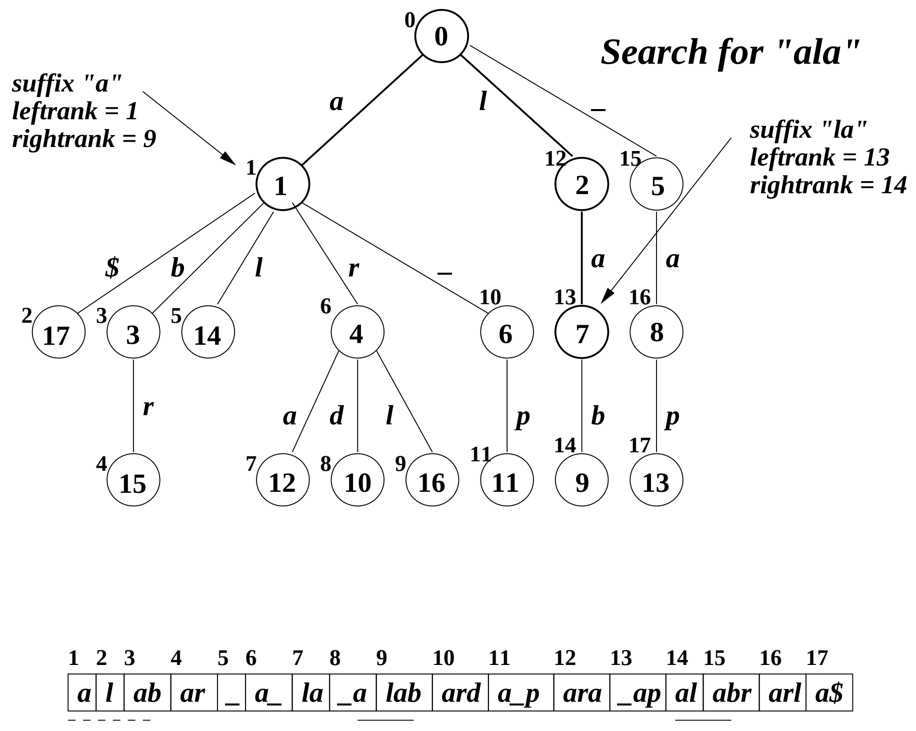


Figure 8: Reporting occurrences of type 2 of $P = ala$ in our running example, first part.

(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 $leftrank_t(Node(S+1)) \in leftrank_t(C_{r+1,m}) \dots rightrank_t(C_{r+1,m})$. 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.

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

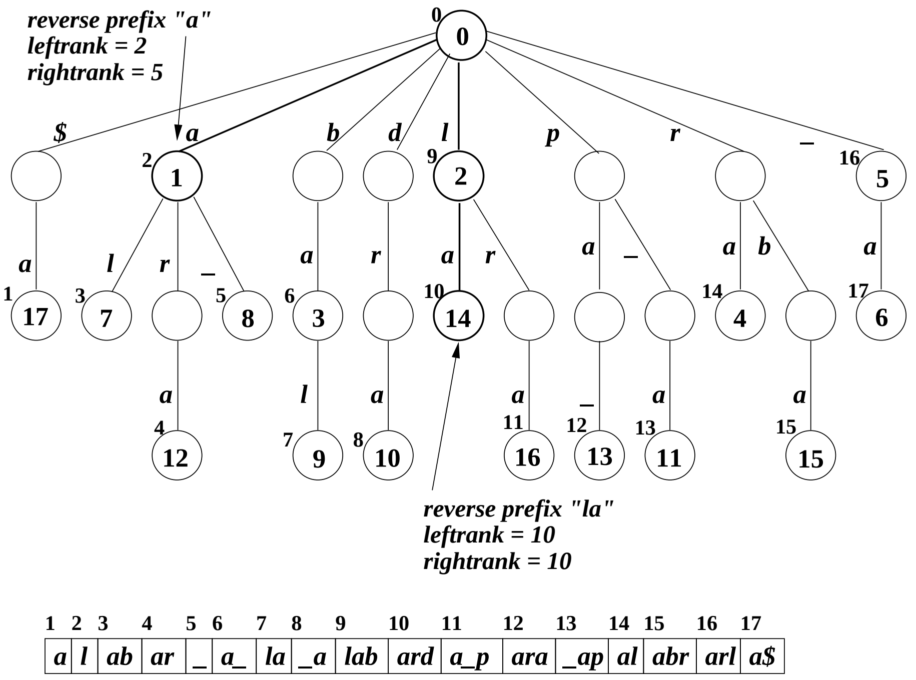


Figure 9: Reporting occurrences of type 2 of $P = ala$ in our running example, second part.

appropriately to the end of the pattern. Three maximal occurrences pass the test, for example $B_1 \dots B_3 = P_{1\dots4}$, 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 $B_9 \dots = P_{2\dots}$, since reading upwards from $Node(8)$ we obtain $P^r_{1\dots1}$.

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 $k \in 0 \dots n$.

4 A Succinct Index Representation

We show now how the data structures used in the algorithm can be implemented using little space.

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

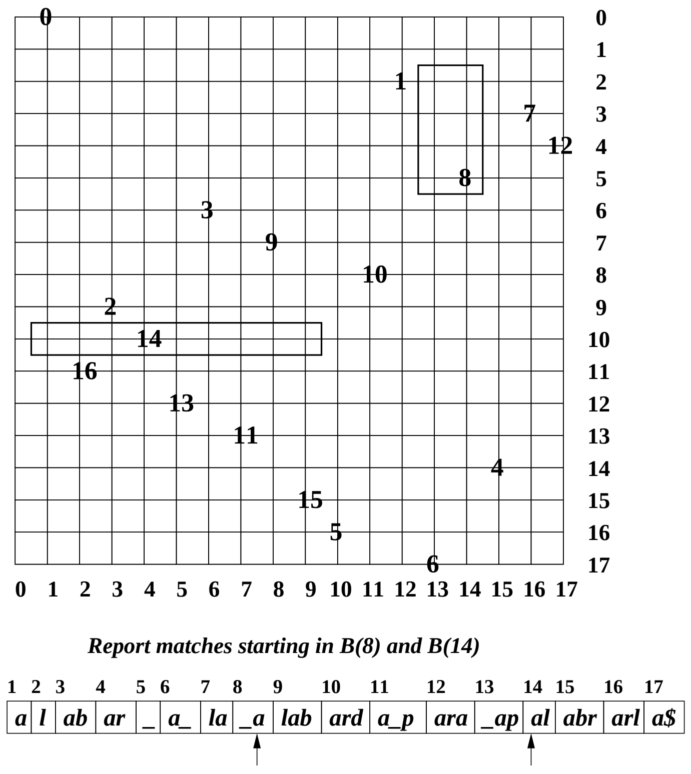


Figure 10: Reporting occurrences of type 2 of $P = ala$ in our running example, third part.

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.

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

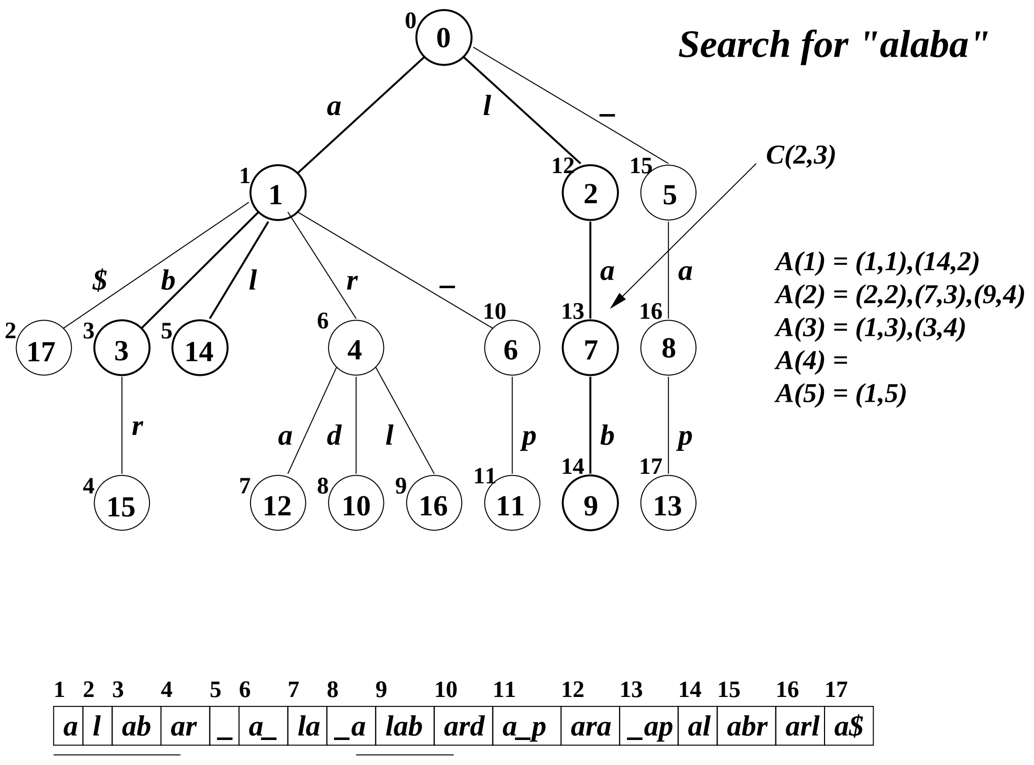


Figure 11: Reporting occurrences of type 3 of $P = alaba$ in our running example, first part.

$2n(1 + \log_2 \sigma) + o(n \log \sigma)$ 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 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

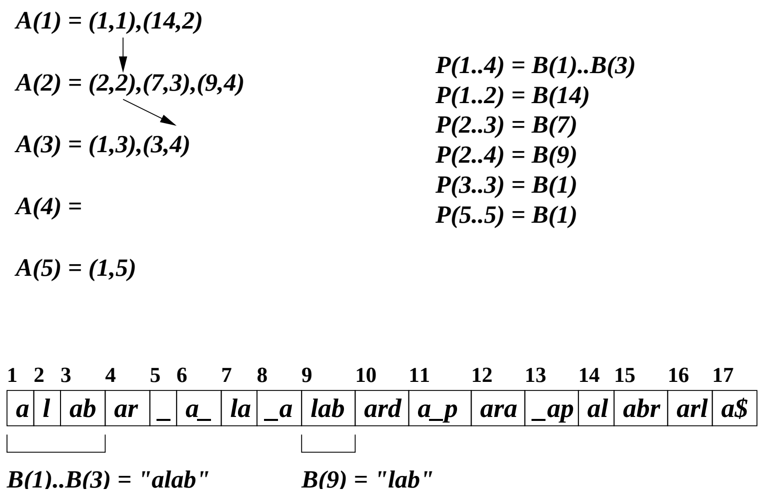


Figure 12: Reporting occurrences of type 3 of $P = alaba$ in our running example, second part.

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 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.

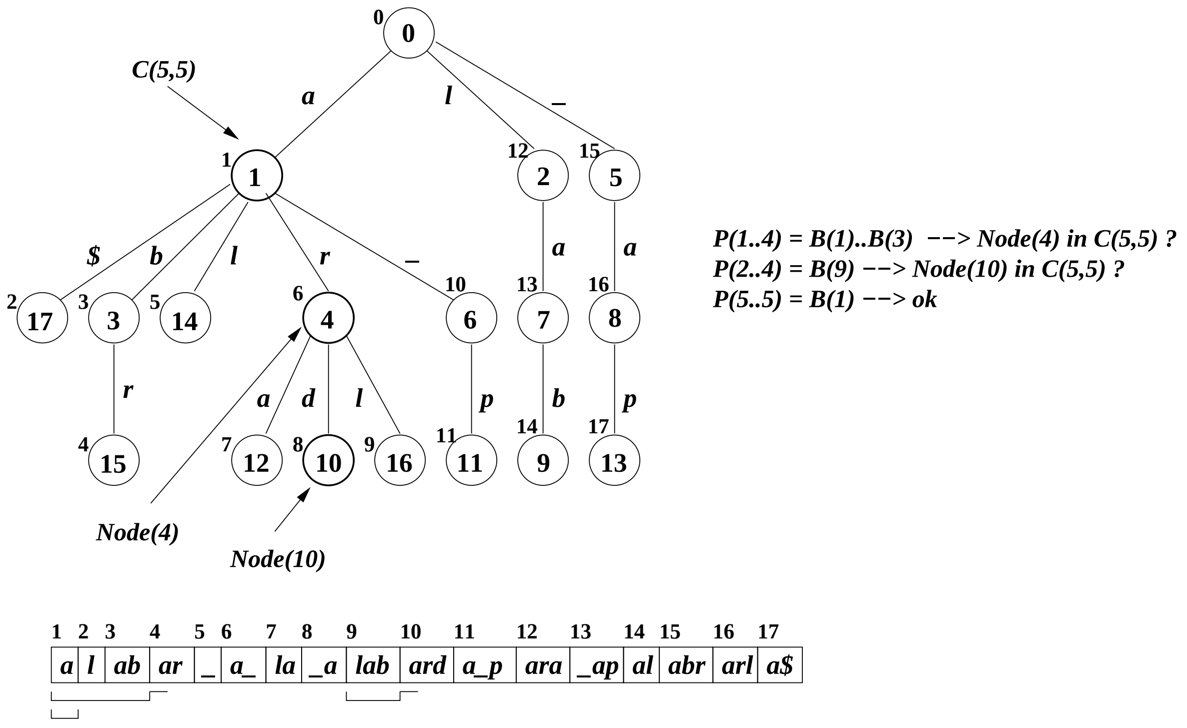

C(5,5)
Node(4)
Node(10)
P(1..4) = B(1)..B(3) --> Node(4) in C(5,5) ?
P(2..4) = B(9) --> Node(10) in C(5,5) ?
P(5..5) = B(1) --> ok
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
a l ab ar _ a_ la _a lab ard a_p ara _ap al abr arl a$

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

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 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 $P_{1...m}$ 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).

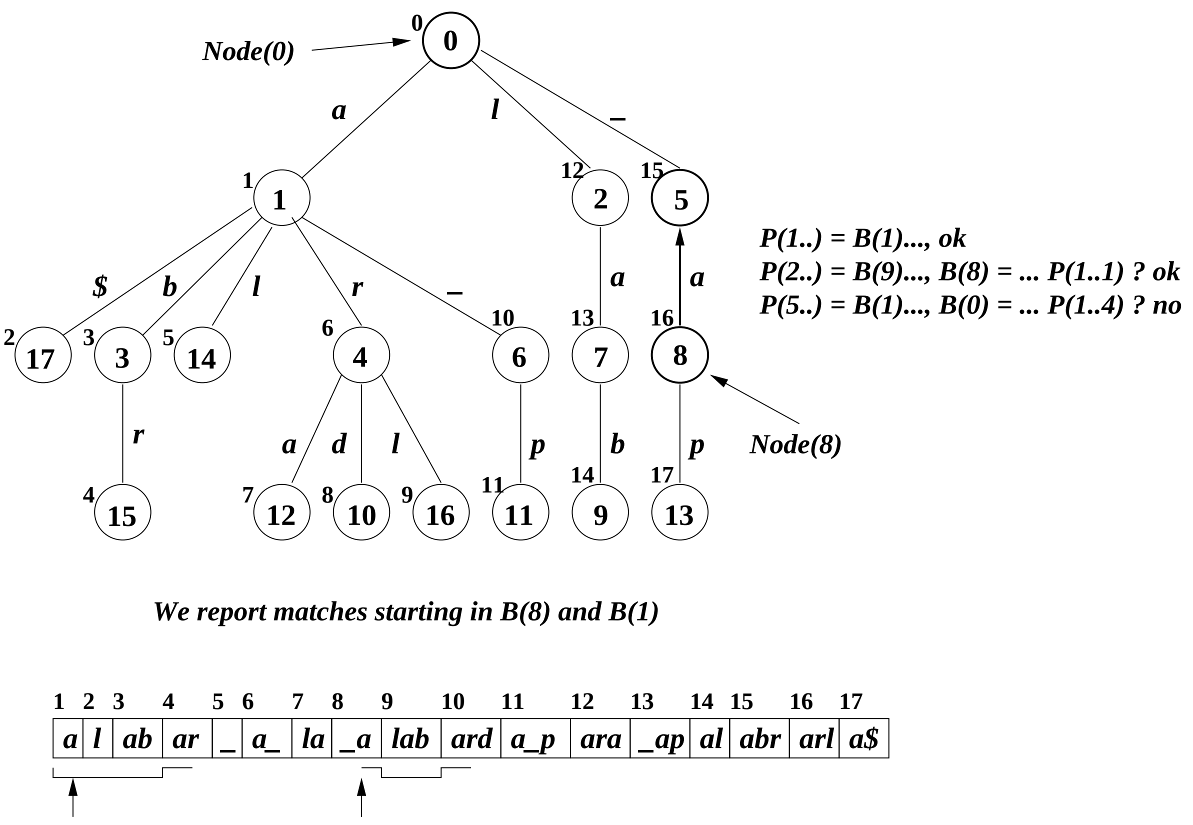

Node(0)
Node(8)
P(1..) = B(1)..., ok
P(2..) = B(9)..., B(8) = ... P(1..1) ? ok
P(5..) = B(1)..., B(0) = ... P(1..4) ? no
We report matches starting in B(8) and B(1)
a l ab ar _ a_ la _a lab ard a_p ara _ap al abr arl a$

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

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 $O(m^2 \log(m\sigma) + m \log n + R \log^{\varepsilon} n)$.

6 Implementation

We briefly describe in this section the implementation of our LZ-index. We focus on the most relevant parts, especially when the theoretically appealing decisions turn out to be difficult to apply in practice. A more detailed discussion of the implementation can be found in [25].

6.1 Balanced Parentheses and General Trees

We represent general trees using a sequence of balanced parentheses, so that each tree node is represented by a couple of matching parentheses. Tree traversal operations are mapped to this sequence, and we seek to support the following operations: $findclose(i)$ finds the position of the

```
**Search** ($P_{1...m}$, $LZTrie$, $RevTrie$, $Node$, $Range$)
 1.                 /* Lying inside a single block */
 2.    $x \leftarrow$ search for $P^r$ in $RevTrie$
 3.    **For** $r \in leftrank_r(x) \ldots rightrank_r(x)$ **Do**
 4.        $y \leftarrow Node(rth_r(r))$
 5.        **For** $z$ in the subtree rooted at $y$ **Do**
 6.            Report $(id_t(z), m + depth(y) - depth(z))$
 7.                 /* Spanning two blocks */
 8.    **For** $i \in 1 \ldots m-1$ **Do**
 9.        $x \leftarrow$ search for $P^r_{1...i}$ in $RevTrie$
10.        $y \leftarrow$ search for $P_{i+1...m}$ in $LZTrie$
11.        Search for $[leftrank_r(x) \ldots rightrank_r(x)]$
                $\times[leftrank_t(y) \ldots rightrank_t(y)]$ in $Range$
12.        **For** $(k, k+1)$ in the result of this search **Do** Report $(k, i)$
13.                 /* Spanning three or more blocks */
14.    **For** $i \in 1 \ldots m$ **Do**
15.        $x \leftarrow$ root node of $LZTrie$
16.        $A_i \leftarrow \emptyset$
17.        **For** $j \in i \ldots m$ **Do**
18.            **If** $x \neq null$ **Then** $x \leftarrow child_t(x, P_j)$
19.            $C_{i,j} \leftarrow x$
20.            $used_{i,j} \leftarrow$ FALSE
21.            **If** $x \neq null$ **Then** $A_i \leftarrow A_i \cup (id_t(x), j)$
22.    **For** $j \in 1 \ldots m$ **Do**
23.        **For** $i \in i \ldots j$ **Do**
24.            **If** $C_{i,j} \neq null$ AND $used_{i,j}$ = FALSE **Then**
25.                $S_0 \leftarrow id_t(C_{i,j})$
26.                $S \leftarrow S_0 - 1,\ r \leftarrow j - 1$
27.                **While** $(S+1, r') \in A_{r+1}$ **Do** /* always exists the 1st time */
28.                    $used_{r+1,r'} \leftarrow$ TRUE
29.                    $r \leftarrow r',\ S \leftarrow S + 1$
30.                $span \leftarrow S - S_0 + 1$
31.                **If** $i > 1$ **Then** $span \leftarrow span + 1$
32.                **If** $r < m$ **Then** $span \leftarrow span + 1$
33.                **If** $span \geq 3$ **Then**
34.                    **If** $C_{r+1,m} = null$ OR
                            $leftrank_t(C_{r+1,m}) \leq leftrank_t(Node(S+1)) \leq rightrank_t(C_{r+1,m})$ **Then**
35.                        $x \leftarrow Node(S_0 - 1),\ i' \leftarrow i - 1$
36.                        **While** $i' > 0$ AND $parent_t(x) \neq null$
                                        AND $x = child(parent_t(x), P_{i'})$ **Do**
37.                            $x \leftarrow parent_t(x),\ i' \leftarrow i' - 1$
38.                        **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.

closing parenthesis that matches opening parenthesis at position i; $parent(i)$ gives the position of the opening parenthesis corresponding to the parent of the node represented by i; and several other simpler ones.

As the solution proposed in [22, 23] to handle balanced parentheses turned out to be too complicated, and the asymptotically vanishing terms turned out to be not so small, we opted for an alternative implementation. It guarantees $O(\log \log n)$ average time for the operation and (almost) guarantees bounded extra space.

Basically, the idea is that, since most trees are small, most of the parentheses sought are close enough in the sequence and could be found after a short brute-force search. For the cases where the answer would not be found, we store the answer directly in a hash table. Hence, only "large" trees have their answer precomputed. In practice the hash tables pose a small space overhead.

The hash tables for the "near" parentheses could be quite large, but we store the distances to the matching parentheses rather than the absolute positions. This reduces the number of bits needed. Collisions are solved because, among all the potential answers that have the same excess (number of opening minus number of closing preceding parentheses), the right answer is the closest one. It is not necessary to store the search key in order to solve collisions.

6.2 LZTrie

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.

The letters and block identifiers corresponding to each node are implemented as simple arrays indexed by rank.

6.3 RevTrie

The reverse 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). In practice the percentage of empty nodes is minimal, and storing them simplifies matters a lot.

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. This is done basically as explained in Section 4. The process is tedious and slow, so we seek to limit it as much as possible. On the other hand, we do not need the $parent(i)$ operation on $RevTrie$.

6.4 Range versus RNode

Instead of implementing the *Range* data structure, we opted by 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[2]). Both for i_t and i_r, we can use *rank* and *rightrank* to determine the ranges in the arrays of block identifiers where the relevant blocks lie. Then we have two choices:

(a) For each block $k+1$ in the block identifiers corresponding to *LZTrie*, ask whether i_r is an ancestor of $RNode(k)$ in *RevTrie* (this operation is easily implemented in a parentheses representation). If so, report block k.

(b) For each block k in the block identifiers corresponding to *RevTrie*, ask whether i_t is an ancestor of $Node(k+1)$ in *LZTrie*. If so, report block k.

Since it is easy to determine which will require less work, we choose the best among both choices. We found that the version based on *RNode* took 1/2 to 2/3 of the time of *Range* for all pattern lengths. Moreover, *RNode* is useful in other points of the search, as we see soon.

6.5 Searching

We search for every pattern substring $P_{i\ldots j}$ 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.

In a second step we search for every reversed pattern prefix, $P^r_{1\ldots j}$, in *RevTrie*, and store 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 $P^r_{1\ldots j}$ and $P_{1\ldots j}$ 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, $P^r_{1\ldots j}$ 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 $P^r_{1\ldots i-1}$ 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).

Occurrences of type 1 and 2 are found as explained. For type 3, instead of the arrays A proposed in the theoretical part, we opt for a hash table where all the triples $(i, j, Cid_{i,j})$ are stored with key $(i, Cid_{i,j})$. 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

[2]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.

involved blocks is $k \ldots k'$, and the pattern area is $i \ldots j'$, then we check that $C_{j'+1,m}$ is an ancestor of $Node(k'+1)$ in $LZTrie$ and that B_{i-1} is an ancestor of $RNode(k-1)$ in $RevTrie$. If all these tests pass, we report block $k-1$.

7 Experimental Results

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.

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.

Our LZ-index takes 1.49 times the text size on ZIFF and 1.19 on DNA. This is 4–5 times the size of the file compressed with Ziv-Lempel, which corroborates our space analysis. We could store the index on disk using less space and quickly reconstruct some parts at load time, but we opt by counting the space the index needs to operate.

We have compared our LZ-index prototype against two of the most prominent alternative proposals. We have considered construction time and space, but our highest interest is in query times, both for counting and for reporting the occurrences.

7.1 Other Indexes Compared

Although our index does not have any relevant space-time tuning parameter, the others do. Hence, we tune the other indexes so as to make them take the same space of our index. The indexes chosen are:

Ferragina and Manzini's FM-index. This index is proposed in [6, 7]. 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 followed rather closely the descriptions in [7] and 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. The main tuning parameter of this index is the sampling step for the suffix array.

Sadakane's CSArray. We obtained from K. Sadakane his implementation of the Compressed Suffix Array index proposed in [27]. We tried different parameter options that gave the same extra space of our index and used those that gave best results.

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

7.2 Comparison

Recall that we compare the three indexes such that they take the same amount of main memory to function. Table 1 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 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 16 shows the overall query times under the different "reporting levels" (just counting the occurrences, reporting their text positions, or showing their text line). 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: our 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. Our rate is close to 900–1,400 per msec. The CSArray is faster than the FM-index at this step, reporting 100–160 occurrences per msec. 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.

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

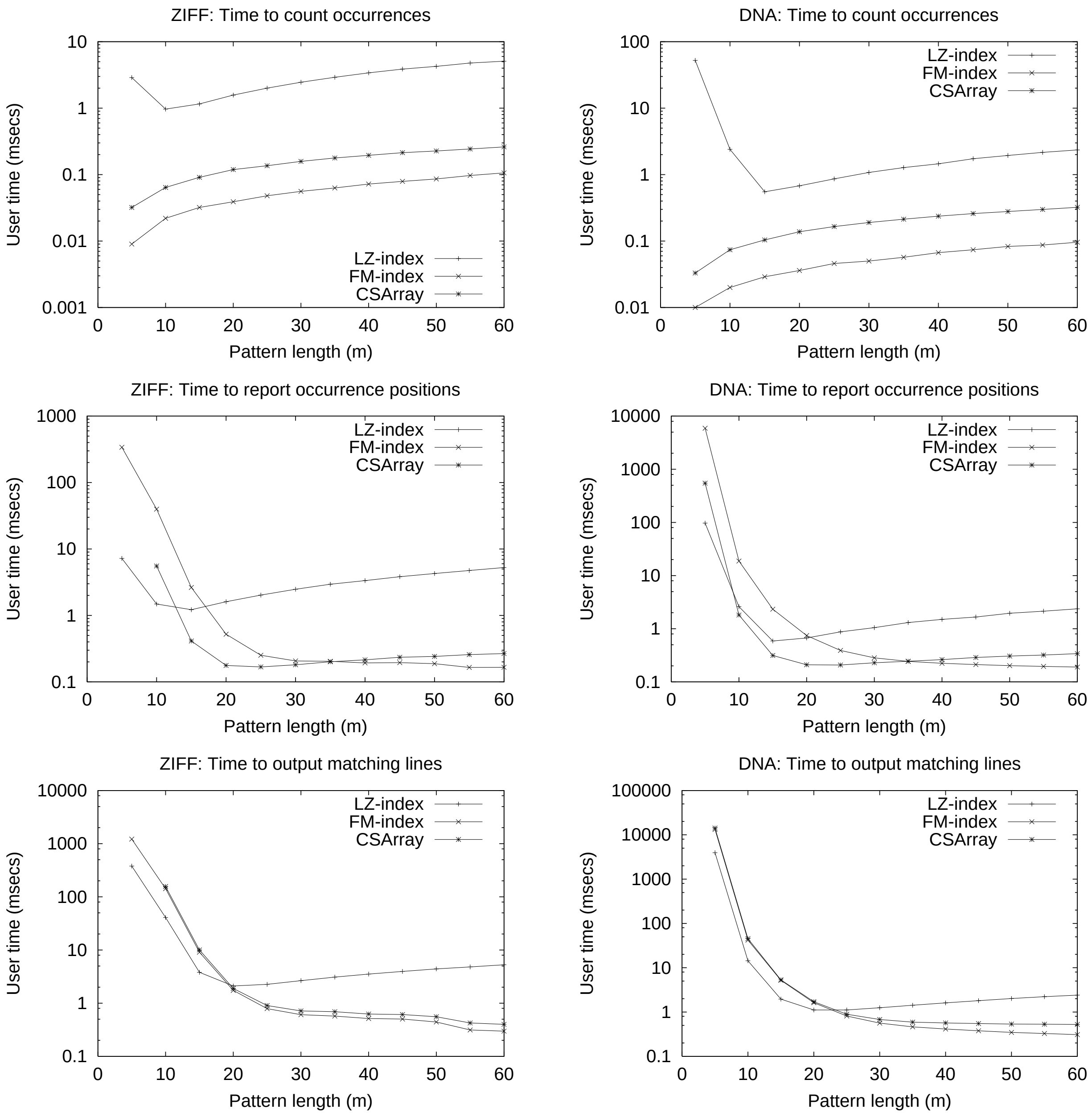


Figure 16: Overall query times when counting occurrences (top), reporting positions (middle), and to output matching lines (bottom). We compare our LZ-index against the most relevant alternatives.

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 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[3]. 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, called the LZ-index. 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 LZ-index and compared our prototype against existing alternatives. The results show that the LZ-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

[3]We believe that the authors have optimized their implementation for a space consumption much inferior than that of our comparison.

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]).

Acknowledgements

We thank Kunihiko Sadakane for kindly giving as a prototype of his index [27].

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. Indexing text using the Ziv-Lempel trie. In *Proc. 9th Intl. Symp. String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 325–336, 2002.

[25] G. Navarro. The LZ-index: A text index based on the Ziv-Lempel trie. Technical Report TR/DCC-2003-1, Dept. of Computer Science, Univ. of Chile, January 2003.

[26] 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.

[27] 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.

[28] 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.

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

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

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