Indexing Text using the Ziv-Lempel Trie

Gonzalo Navarro *

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 that it is possible to build a data structure based on the Ziv-Lempel trie that takes $4n \log_2 n(1+o(1))$ bits of space and reports the R occurrences of a pattern of length m in worst case time $O(m^2 \log(m\sigma) + (m+R)\log n)$.

1 Introduction

Modern text databases have to face two opposed goals. On 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 succint form, yet permitting efficient text searching.

Let our text $T_{1...u}$ be a sequence of characters over an alphabet Σ of size σ, and let the search pattern $P_{1...m}$ be another (short) sequence over Σ. Then the text search problem consists in finding all the occurrences of P in T.

Despite that there has been some work on succint inverted indexes for natural language for a while [22,19], until a short time ago it was believed that any general index for string matching would need $\Omega(u)$ space. In practice, the smaller indexes available were the suffix arrays [15], 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).

In the last decade, several attempts to reduce the space of the suffix trees [2] or arrays have been made by Kärkkäinen and Ukkonen [9,12], Kurtz [13] and Mäkinen [14], obtaining reasonable improvements, albeit no spectacular ones (at best 9 times the text size). Moreover, they have concentrated on the space requirement of the data structure only, needing the text separately available.

Grossi and Vitter [7] 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 apart from the index in order to answer queries.

Following this line, Sadakane [20] presented a suffix array implementation 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 [5] 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.

* Dept. of Computer Science, Univ. of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

(In a real implementation [6] they removed this constant at the price of a not guaranteed search time.)

However, there are older attempts to produce succint indexes, by Kärkkäinen and Ukkonen [11,10]. 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 [10] 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. 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 $O(m^2\log(m\sigma)+(m+R)\log n)$. It is well known that Ziv-Lempel compression asymptotically approaches the true entropy H of the text [3]. Since the compressed text needs at least $n\log_2 n$ bits of storage, we have that our index is opportunistic, taking at most $4uH$ bits. There are no large constants involved in the sublinear part.

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 $O(m^2\log(m\sigma)+m\log n+R\log^{\varepsilon}n)$ provided we pay $O\left(\frac{1}{\varepsilon}n\log n\right)$ space.

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 [3]. We are particularly interested in the LZ78/LZW format, which we describe in depth.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [23]) 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 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 give as an example the compression of the word *ananas* in Figure 1. The first block is $(0,a)$, and next $(0,n)$. When we read the next a, a is already the block 1 in the dictionary, but an is not in the dictionary. So we create a third block $(1,n)$. We then read the next a, a is already the block 1 in the dictionary, but as does not appear. So we create a new block $(1,s)$.

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

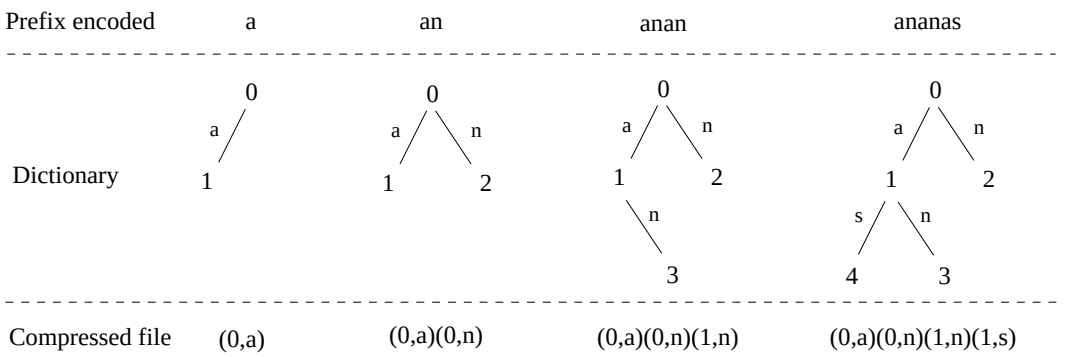

Fig. 1. Compression of the word *ananas* with the algorithm LZ78.

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 [21]. 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 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 $B_0 = \varepsilon$; $\forall k \neq \ell,\ B_k \neq B_\ell$; and $\forall k \geq 1,\ \exists \ell < k, c \in \Sigma,\ B_k = B_\ell \cdot c$.

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 succint representation, and consequently the space occupancy and time complexity, is considered in the next section.

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 r-th string in $B_0 \dots B_n$ in lexicographical order.

2. $RevTrie$: is the trie formed by all the reverse strings $B_0^r \dots B_n^r$. 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.
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 $[0 \dots n] \times [0 \dots n]$. The points stored in this structure are $\{(revrank(B_k^r), rank(B_{k+1})),\ k \in 0 \dots n-1\}$, where $revrank$ is the lexicographical rank in $B_0^r \dots B_n^r$ and $rank$ is the lexicographical rank in $B_0 \dots B_n$.

3.2 Search Algorithm

Let us now consider the search process. We distinguish three types of occurrences of P in T, depending on the block layout (see Figure 2):

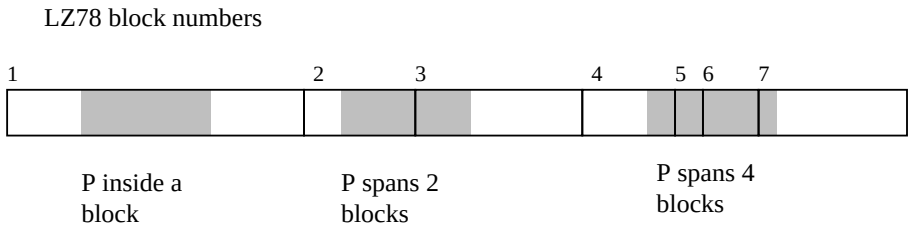

Fig. 2. Different situations in which P can match inside T.

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

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 them several times, since we will report a subtree that is contained in another subtree reported. To avoid this we keep track of the last m characters that the current node represents. When this string equals P, we have arrived at another node that has been or will be reported elsewhere so we stop that branch. The equality condition can be tested in constant time using a KMP-like algorithm.

Occurrences spanning two blocks.

We do not know the position where P has been split, 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.
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.

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 amount 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 $P_{i\ldots}$. 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 $P_{i\ldots j} = B_k \ldots B_\ell$ contained in the pattern, we determine whether B_{k-1} finishes with $P_{1\ldots i-1}$ and $B_{\ell+1}$ starts with $P_{j+1\ldots m}$. 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 $P_{i\ldots j}$ 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 $P_{i\ldots j}$ we store a null value in $C_{i,j}$ and don't modify A_i. At the end of every i-turn, we sort A_i by block number. Mark every $C_{i,j}$ as *unused*.
2. For every $1 \le i \le j < m$, for increasing j, try to extend the match of $P_{i\ldots j}$ 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 already tried 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 $P_{i\ldots r}$ found ending at block S, check whether block $S+1$ starts with $P_{r+1\ldots m}$, 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.

(b) If block $S+1$ starts with $P_{r+1\ldots m}$, check whether block S_0-1 finishes with $P_{1\ldots i-1}$. 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 $P^r_{1\ldots i-1}$.

(c) If this is the case, 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 3.2 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 can also 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 $k \in 0 \ldots n$.

4 A Succint 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 [17] 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. [18] 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 [18] 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 $2n(1+\log_2 \sigma) + o(n \log \sigma)$ 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.

```
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 not containing P again 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 AND C_{r+1,m} ≠ null Then
                    If leftrank_t(Node(S + 1)) ∈ leftrank_t(C_{r+1,m}) ... 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)
```

Fig. 3. The search algorithm. The value $depth(y) - depth(z)$ is determined on the fly since we traverse all the subtree of z.

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 leafs 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. Searching in $RevTrie$ has the same cost as in $LZTrie$.

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 $LZTrie$: given the target node x of an edge we want to traverse, we obtain using $Node(rth_r(leftrank_r(x)))$ a node in $LZTrie$ that represents a binary string whose (reversed) suffix matches the edge we want to traverse. Then, we use the $parent_t$ pointers to read upwards the (reverse) string associated to the block in the reverse trie.

For the $Node$ mapping we simply pay 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 [1,10], we prefer one of minimal space requirement by Chazelle [4]. 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 [8,16] needs $N + o(N)$ bits to represent a bitmap of length N, and permits the $rank$ operation 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 $n\lceil \log_2 n \rceil (4 + o(1))$. The $o(1)$ term does not hide large constants, just $\frac{5+2\log_2 \sigma + 2\log_2 \log_2 n}{\log_2 n} + o(1/\log n)$. The tries and $Node$ can be built in $O(u \log \sigma)$ time, while $Range$ needs $O(n \log n)$ construction time. Since $n \log n = O(u \log \sigma)$ [3], 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 $O(m \log \sigma)$. 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 $O(m^2 \log \sigma)$, 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 26–28, 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.

Hence the total search cost to report the R occurrences of pattern $P_{1...m}$ is $O(m^2 \log(m\sigma) + (m+R)\log n)$. If we consider the alphabet size as constant then the algorithm is $O(m^2 \log m + (m+R)\log n)$. The existence problem can be solved in $O(m^2 \log(m\sigma) + m \log n)$ 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 $T_{1...u}$ in $O(u \log \sigma)$ time.

Chazelle [4] permits several space-time tradeoffs in his data structure. In particular, by paying $O\left(\frac{1}{\epsilon} n \log n\right)$ space, reporting time can be reduced to $O(\log^\epsilon n)$. If we pay for this space complexity, then our search time becomes $O(m^2 \log(m\sigma) + m \log n + R \log^\epsilon n)$.

6 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 $O(m^2 \log(m\sigma) + (m+R)\log n)$ time.

Future work involves obtaining a real implementation of this index. Some numerical exercises show that the index should be practical. For example, assume a typical English text of 1 Mb, which is compressed by *Unix's Compress* to about 1/3 of its size. Given the space used by this program to code each block, we have that there are about $n \approx u/10$ blocks. Our index needs $4n \log_2 n(1 + o(1)) \approx 9.7u$ bits, little more than the size of the uncompressed text ($8u$ bits in ASCII). This should stabilize for longer texts: the 11-th order entropy of English text has been found to be 2.4 bits per symbol [3], and our index takes under this model $4uH_{11} = 9.6u$ bits of space. It is estimated [3] that the true entropy H of English text is around 1.3 bits per symbol (considering orders of 100 or more). Under this model our index takes $4uH = 5.2u$ bits, smaller than the uncompressed text. Note that in this space we also store the compressed representation of the text.

References

1. P. Agarwal and J. Erickson. Geometric range searching and its relatives. *Contemporary Mathematics*, 23: Advances in Discrete and Computational Geometry:1–56, 1999.
2. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
3. T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.
4. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
5. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symp. Foundations of Computer Science (FOCS'2000)*, pages 390–398, 2000.
6. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th ACM Symp. on Discrete Algorithms (SODA'2001)*, pages 269–278, 2001.
7. 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'2000)*, pages 397–406, 2000.
8. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS'89)*, pages 549–554, 1989.
9. 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.
10. 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.
11. 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.
12. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Ann. Intl. Conference on Computing and Combinatorics (COCOON'96)*, LNCS 1090, 1996.

13. S. Kurtz. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Universität Bielefeld, 1998.
14. V. Mäkinen. Compact suffix array. In *Proc. 11th Ann. Symp. Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 305–319, 2000.
15. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993. Earlier version in *Proc. SODA'90*.
16. I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, LNCS 1180, pages 37–42, 1996.
17. 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.
18. I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. In *Proc. 18th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'98)*, LNCS 1530, pages 186–195, 1998.
19. 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.
20. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Intl. Symp. Algorithms and Computation (ISAAC'2000)*, LNCS 1969, pages 410–421, 2000.
21. T. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.
22. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, second edition, 1999.
23. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. on Information Theory*, 24:530–536, 1978.