Implementing the LZ-index: Theory versus Practice

1.2

GONZALO NAVARRO
University of Chile

The LZ-index is a theoretical proposal of a lightweight data structure for text indexing, based on the Ziv-Lempel trie. If a text of u characters over an alphabet of size σ is compressible to n symbols using the LZ78 algorithm, then the LZ-index 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 $O(m^3 \log \sigma + (m + R) \log n)$. In this paper we face the challenge of obtaining a practical implementation of the LZ-index, which is not at all straightforward from the theoretical proposal. We end up with a prototype that takes the promised space and has average search time $O(\sigma m \log u + \sqrt{uR})$. This prototype is shown to be faster than other competing approaches when we take into account the time to report the positions or text contexts of the occurrences found. We show in detail the process of implementing the index, which involves interesting lessons of theory versus practice.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and problems—*Pattern matching, computations on discrete structures, sorting and searching*; H.2.1 [**Database Management**]: Physical design—*Access methods*; H.3.2 [**Information Storage and Retrieval**]: Information storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Data structures, data storage representations, coding and information theory, indexing methods, textual databases

ACM Reference Format:

Navarro, G. 2008. Implementing the LZ-index: Theory versus practice. ACM J. Exp. Algor. 13, Article 1.2 (November 2008), 49 pages. DOI = 10.1145/1412228.1412230 http://doi.acm.org/10.1145/1412228.1412230

1. INTRODUCTION AND RELATED WORK

A *text database* is a system providing fast access to a large mass of textual data. By far its most challenging requirement is that of performing fast text

Supported in part by Fondecyt Grant 1-080019, Chile.
Authors' address: Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile; email: gnavarro@dcc.uchile.cl.

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. The text search problem is that 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 hand, they have to use as little space as possible. The goals are opposed because, to provide fast access, an *index* has to be built on the text. This index is a data structure 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 [Witten et al. 1999; Navarro et al. 2000] (able of finding whole words and phrases), until a short time ago it was believed that any general index for string matching would need much more space. In practice, the smallest indexes available were the suffix arrays [Manber and Myers 1993], requiring $u \log_2 u$ bits to index a text of u characters. Since the text requires $u \log_2 \sigma$ bits to be represented, this index is usually much larger than the text (typically 4 times the text size).

Since the last decade, several attempts to reduce the space of the suffix trees [Apostolico 1985] or arrays have been made by Kärkkäinen [1995], Kurtz [1998], Mäkinen [2003], and Abouelhoda et al. [2002]. These approaches have been mainly practical, in the sense that it was easy to obtain an implementation from the algorithmic formulation. The results have been remarkable, but not spectacular.

A parallel, much more theoretical track, started at about the same time, thanks to Kärkkäinen and Ukkonen [1996a, 1996b], Kärkkäinen [1999], Grossi and Vitter [2000], Sadakane [2000, 2002], and Ferragina and Manzini [2000, 2001, 2002]. These results came together with encouraging analytical results on the performance of the indexes. In particular, in the works of Sadakane and Ferragina and Manzini, the indexes *replace* the text and, using little space (sometimes even less than the original text), provide indexed access. This is an unprecedented breakthrough. On the other hand, many of the proposals in this track have never been implemented. Those that were implemented (e.g., that of Ferragina and Manzini [2001]), needed important changes in their formulation to become practical. In these changes, several of the theoretical guarantees were traded for a workable approach. The result has been practical prototypes that mostly matched the theoretical promises.

Hence, we have a class of algorithms and data structures that promise a drastic reduction in the space overhead of text indexes. On the other hand, they provide an excellent example of the challenges faced to implement a theoretical proposal.

We proposed a new index on these lines [Navarro 2002, 2004], called the LZ-index. It is based on the Ziv-Lempel parsing of the text. The theoretical

proposal shows that, if the Ziv-Lempel parsing cuts the text into n phrases, then the index takes $4n \log_2 n(1 + o(1))$ bits of space, which is 4 times the size of the compressed text and also 4 times the text entropy [Kosaraju and Manzini 1999; Ferragina and Manzini 2002]. The LZ-index answers queries in $O(m^3 \log \sigma + (m + R) \log n)$ worst case time. The index also replaces the text: It can reproduce a text context of length L (formed by whole phrases) in $O(L \log \sigma)$ time, or obtain the whole text in time $O(u \log \sigma)$. The index is built in $O(u \log \sigma)$ time.

The main goal of this article is to show how a practical implementation of the LZ-index is obtained. The implementation involves several considerations on the practicality of the theoretical decisions, which offer worst-case big-O guarantees but do not care for the constants, the memory hiearchy, the simplicity of the solutions, and several other important practical aspects. These decisions are considered in detail, and several lessons of theory versus practice come out in the way.

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). On a 2GHz Pentium IV machine, the index is built at a rate of 1 to 2 megabytes (MB)/second (which is competitive with current technology) and uses a temporary extra space similar to a suffix array construction (five times the text size, which is large but usual, and can be reduced in 50% by standard means). On a 50 megabytes 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 to 800 per msec. Text lines can be displayed at a rate of 14 lines per msec.

We have compared our index against existing alternatives [Sadakane 2000; Ferragina and Manzini 2000]. 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 to 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 to 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 the index to reproduce it.

A prototype of our index is available at `http://www.dcc.uchile.cl/gnavarro/software`.

This article is organized as follows: In Section 2, we explain the Ziv-Lempel compression. In Section 3, to make the article self-contained, we recall the basic ideas behind the LZ-index. Similarly, Section 4 reviews the theoretical proposal to represent the data structures in succinct space and the corresponding analysis. From then on, we switch to the practical part of the article. Sections 5 through 10 describe the actual implementation of the different components of the index. Section 11 considers index construction and space, and Section 12 shows how to execute queries. Section 13 compares the implementation against the most prominent alternatives. Finally, Section 14 gives our conclusions and future work directions.

Experimental setup. To demonstrate the results in practice, we have chosen two different text collections. The first, ZIFF, contains 83.37MB obtained from the "ZIFF-2" disk of the TREC-3 collection [Harman 1995]. The second, DNA, contains 51.48MB from *GenBank* (Homo Sapiens DNA, `http://www.ncbi.nlm.nih.gov`), with lines cut every 60 characters. We use the whole collections as well as incremental subsets of them.

Our tests have been run on a Pentium IV processor at 2GHz, 512MB 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 in main memory, we measure CPU times.

The search patterns were obtained by pruning text lines to their first m characters. In the case of DNA, we avoided patterns with five or more "N" characters, which represents an unknown character and is not searchable. For ZIFF, we avoided lines containing tags and invisible characters like "&."

Historical note. This article was accepted in 2003. For clerical reasons, its publication has been long delayed. In between, much progress has been made in the theory and practice on compressed text indexes. A recent article [Navarro and Mäkinen 2007] surveys most of the theoretical advances and, in particular, serves as an update of the account of existing indexes given here. As for the practical advances, we point to the *Pizza&Chili* site, at `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`, where most of the relevant theoretical proposals have been implemented under a standardized interface that simplifies comparisons, as well as standard texts of different types. The implementation described in this article is now in the site,[1] as well as several more recent versions of it.

Despite being somewhat outdated, this article is still worthy. First, it illustrates the theory/practice tradeoffs made and lessons learned when implementing a highly theoretical proposal. Second, it is the only place where a detailed description of the implementation of the LZ-index is available, and this is today among the most practical compressed text indexes. Third, its conclusions about the suitability of the LZ-index versus alternative compressed indexes are still valid, as can be witnessed by comparisons carried out on the *Pizza&Chili* site and informed on a paper available in there.

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, Bell et al. [1990]. We are particularly interested in the LZ78 format, which we describe in depth.

[1]To comply with the standard interface, the capability of reporting exact text positions was added to the *Pizza&Chili* version, whereas `http://www.dcc.uchile.cl/gnavarro/software` contains the original one described in this paper.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [Ziv and Lempel 1978]) is based on a dictionary of blocks (or "phrases") 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 $T_{1...j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \dots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1...u}$ 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 $b_{r+1} = (s, T_{j+\ell_s+1})$, write the pair at the end of the compressed file Z (i.e., $Z = b_1 \dots b_r b_{r+1}$) and add the block to the dictionary. It is easy to see that this dictionary is prefix-closed (that is, any prefix of an element is also an element of the dictionary) and a natural way to represent it is a trie.

We will call B_i the string represented by block b_i; thus, $B_{r+1} = B_s T_{j+\ell_s+1}$ and $T = B_0 \dots B_n$. Also, let $b_r = (r_1, c_1)$, $b_{r_1} = (r_2, c_2)$, $b_{r_2} = (r_3, c_3)$ and so on until $r_k = 0$. The sequence r, r_1, r_2, $\dots$ is called the *referencing chain* starting at block r. It reproduces the way block b_r is formed from previous blocks and is obtained by successively moving to the parent in the dictionary trie.

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 and considered to be smaller than any other character) at the end of the text. The last block will contain this character and thus will be unique too.

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 basically deal with the best way to code the pairs in the compressed file. A coding variant called LZW [Welch 1984] is implemented in Unix's *Compress* program.

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. THE BASICS OF THE LZ-INDEX

We now review [Navarro 2004] the basic idea to search for a pattern $P_{1...m}$ a text $T_{1...u}$ that has been compressed using the LZ78 algorithm into $n+1$ blocks $T = B_0 \dots B_n$, such that $B_0 = \varepsilon$; $\forall k \neq \ell,\ B_k \neq B_\ell$ (i.e., no two blocks are equal); and $\forall k \geq 1,\ \exists \ell < k, c \in \Sigma,\ B_k = B_\ell \cdot c$ (i.e., every block except B_0 is formed by a previous block plus a letter at the end).

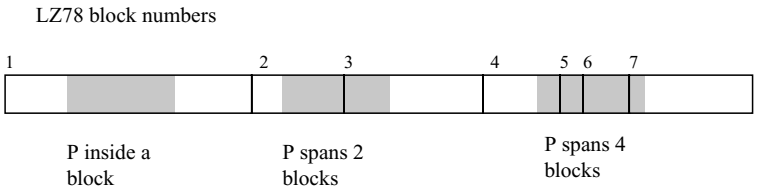

Fig. 1. Different situations in which P can match inside T.

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 subsequent sections.

1. *LZTrie*: is the trie formed by all the blocks $B_0 \dots B_n$. 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 $B_0 \dots B_n$;
   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 yields the block identifier representing the lexicographically r-th string of $\{B_0 \dots B_n\}$.
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 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 $\{B_0 \dots B_n\}$. For each such point, the corresponding k value is stored.

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 (Figure 1):

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

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

3.2.1 *Occurrences Lying Inside a Single Block.* Given the properties of LZ78, 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*.

Therefore, to detect all the occurrences that lie inside a single block, we do the following:

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 $r \in leftrank_r(x) \dots rightrank_r(x)$, 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 subtrees that are contained in other subtrees reported.

3.2.2 *Occurrences Spanning Two Blocks.* P can be split at any position, so we have to try all of them. 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 $i \in 1 \dots m-1$, split P into $pref = P_{1...i}$ and $suff = P_{i+1...m}$ 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 $[leftrank_r(x) \dots rightrank_r(x)] \times [leftrank_t(y) \dots rightrank_t(y)]$ 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.

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

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

1. For every $1 \le i \le j \le m$, search for $P_{i...j}$ in *LZTrie* and record the node x found in $C_{i,j} = x$, as well as add $(id_t(x), j)$ to array A_i. The search is made for increasing i and for each i value we increase j. This way, we perform a single search in the trie for each i. If there is no node corresponding to $P_{i...j}$, we stop searching and adding entries to A_i, and store null values in $C_{i,j'}$ for $j' \ge j$. At the end of every i-turn, A_i is already sorted 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 $P_{i...j}$ to the right. We do not extend to the left because this, if useful, has been done already (we mark used ranges to avoid working on a sequence that has been tried already from the left). Let S and S_0 denote $id_t(C_{i,j})$ and find $(S+1, r)$ in A_{j+1}. If r exists, mark $C_{j+1,r}$ as *used*, increment S and repeat the process from $j = r$. Stop when the occurrence cannot be extended further (no such r is found).
   a. For each maximal occurrence $P_{i...r}$ found ending at block S such that $r < m$, check whether block $S+1$ starts with $P_{r+1...m}$, i.e., whether $leftrank_t(Node(S+1)) \in leftrank_t(C_{r+1,m}) \dots rightrank_t(C_{r+1,m})$. 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 must make sure that the occurrences reported span at least 3 blocks.

Figure 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 from the beginning of the occurrence to 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. 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 THEORETICAL IMPLEMENTATION PROPOSAL

To make this article self-contained and to permit contrasting the practical implementation decisions against the theoretical proposal [Navarro 2004] to implement this index obtaining worst case guarantees, we now review this theoretical proposal.

4.1 A Succinct Index Representation

Let us first consider the tries. Munro and Raman [1997] show how 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. [2001] 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 article, Munro et al. [2001] show that a trie can be represented using this same structure by expressing 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 of a character 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,

```
Search (P_{1...m}, LZTrie, RevTrie, Node, Range)
1.                  /* Lying inside a single block */
2.      x ← search for P^r in RevTrie
3.      For r ∈ leftrank_r(x) ... rightrank_r(x) Do
4.          y ← 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 ∈ 1 ... m − 1 Do
9.          x ← search for P^r_{1...i} in RevTrie
10.         y ← search for P_{i+1...m} in LZTrie
11.         Search for [leftrank_r(x) ... rightrank_r(x)]
                    ×[leftrank_t(y) ... 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 ∈ 1 ... m Do
15.         x ← root node of LZTrie
16.         A_i ← ∅
17.         For j ∈ i ... m Do
18.             If x ≠ null Then x ← child_t(x, P_j)
19.             C_{i,j} ← x
20.             used_{i,j} ← FALSE
21.             If x ≠ null Then A_i ← A_i ∪ (id_t(x), j)
22.     For j ∈ 1 ... m Do
23.         For i ∈ 1 ... j Do
24.             If C_{i,j} ≠ null AND used_{i,j} = FALSE Then
25.                 S_0 ← id_t(C_{i,j})
26.                 S ← S_0 − 1, r ← j − 1
27.                 While (S + 1, r') ∈ A_{r+1} Do /* always exists the 1st time */
28.                     used_{r+1,r'} ← TRUE
29.                     r ← r', S ← S + 1
30.                 span ← S − S_0 + 1
31.                 If i > 1 Then span ← span + 1
32.                 If r < m Then span ← span + 1
33.                 If span ≥ 3 Then
34.                     If C_{r+1,m} = null OR (leftrank_t(C_{r+1,m}) ≤
                            ≤ leftrank_t(Node(S + 1)) ≤ rightrank_t(C_{r+1,m})) Then
35.                         x ← Node(S_0 − 1), i' ← i − 1
36.                         While i' > 0 AND parent_t(x) ≠ null
                                        AND x = child(parent_t(x), P_{i'}) Do
37.                             x ← parent_t(x), i' ← i' − 1
38.                         If i' = 0 Then Report (S_0 − 1, i − 1)
```

Fig. 2. The search algorithm. The value $depth(y) - depth(z)$ is determined on the fly since we traverse the whole subtree of z.

we use the extra bit allocated: at every true node that happens to be internal, we add a leaf by the bit 0, while all the other children necessarily descend by the bit 1.

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

The operations $parent_t$ and $child_t$ can, therefore, be implemented in $O(\log \sigma)$ time. The remaining operations, $leftrank_t(x)$ and $rightrank_t(x)$, 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 also use 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 be explained how we store the representation of the strings in the reverse trie, since 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 node y 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 $Node(rth_r(leftrank_r(x))$ and $Node(rth_r(rightrank_r(x))$, two nodes in *LZTrie*. We have to go up from both nodes until we read a^r (string a reversed), and then we continue going up to the parent in *LZTrie*. What we read after a^r is b^r. The process finishes when the characters read from both nodes differ 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 [Agarwal and Erickson 1999; Kärkkäinen 1999], we prefer one of minimal space requirement by Chazelle [1988]. This structure is a perfect binary tree dividing the points along one coordinate plus a 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 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 $[0, n] \times [0, n]$, answering queries in $O((R+1) \log n)$ time, where R is the number of occurrences reported. A newer technique for bitmaps [Jacobson 1989; Munro 1996] needs $N + o(N)$ bits to represent a bitmap of length N and executes the $rank$ operation (here 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).

4.2 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 $n \log n = O(u \log \sigma)$ [Bell et al. 1990], the overall construction time is $O(u \log \sigma)$. Let us now consider the search time of the algorithm.

Finding the blocks that totally contain P requires a search in *RevTrie* of cost $O(m^2 \log \sigma)$. Later, we collect occurrences: for each unit of work done, 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 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 through 29 in Figure 2, 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 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.

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). Finally, we can uncompress and show a 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 [1988] permits several space-time tradeoffs in his data structure. In particular, by paying $O(\frac{1}{\varepsilon} n \log n)$ space, reporting time can be reduced to $O(\log^{\varepsilon} n)$. If we pay for this space complexity, then our search time becomes $O(m^3 \log \sigma + m \log n + R \log^{\varepsilon} n)$.

5. IMPLEMENTING BITMAPS

One of the lowest level data structures is the bitmap able to answer the query *rank(i)*, which is the number of 1's before position i. Jacobson [1989], and later Munro [1996], showed how to use $\ell + o(\ell)$ bits to represent a bitmap of length ℓ, implementing *rank* in constant time.

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

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

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

where a computer word of 32 bits is assumed, but it can be trivially extended to 64 bits at almost the same cost. This solution requires 13 operations, all on a single register. This is usually faster than a single access to RAM memory.[3]

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

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

Our above example with 64 megabits, using $w = 32$, would be as follows: $s = 32 \times 26 = 832$, hence we need $\lceil \log_2 s \rceil = 10$ bits per block counter, and redefine $s = 1024$. We need only 33.79% of extra space. Figure 3 (right) shows, among other things, the experimental space overhead of the bitmaps in our examples. It ranges between 33.30% and 33.60% as expected.

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

---

[2]We have taken all the constants literally, without considering that those can be changed to reduce the space while maintaining the same theoretical time complexity. Yet, our aim is to illustrate that lower-order terms should be considered seriously.

[3]In 2005, we carried out a more thorough study and found other combinations that gave us slightly better performance [González et al. 2005]. We also found variants that performed decently while using just 5% of extra space. However, as bitmaps are not a large part in the space consumption of our LZ-index, we stick to the larger and faster variant.

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

6. IMPLEMENTING BALANCED PARENTHESES

The proposals [Munro and Raman 1997; Munro et al. 2001] to handle trees and tries in succinct space build on top of a succinct representation of a sequence of balanced parentheses. As we follow the same path, we study now in depth a practical implementation, keeping in mind that our goal is to represent a general tree in preorder (ancestors enclose their descendants).

The solution [Munro and Raman 1997] to store a sequence of p parentheses uses $o(p)$ extra space and permits executing the following operations in constant time: $findopen(i)$ and $findclose(i)$ find the position of the opening(closing) parenthesis that matches closing(opening) parenthesis at position i; $excess(i)$ gives the excess of opening parentheses over closing parentheses up to position i; and $enclose(i)$ gives the opening position of the closest parentheses pair that encloses opening parenthesis i.

In practice, the solution [Munro and Raman 1997] is overwhelmingly complicated, involving a complex structure that is replicated at three levels (big, small, and tiny blocks). Which is worse, the extra "sublinear" space is extremely large in practice. The authors mention that the sequence should have more than 20 million bits before the excess becomes less than 100%. We made ourselves the exercise of adding up all the sublinear-size data structures for our 64 megabit example, and it turned out that the extra space is 15 times p. Moreover, we did not count a precomputed table whose size is $2^{4(\log_2 \log_2 p)^2} 8(\log_2 \log_2 p)^2(2 + 2 \log_2 \log_2 \log_2 p) \approx 2^{111}$ bits.[4]

As it is clear that we cannot go ahead with this approach in practice, we design a simpler scheme that guarantees $O(\log \log p)$ average time and (almost) guarantees bounded extra space. We do reuse some of the ideas of Munro and Raman [1997], but we combine them with practical considerations.[5]

6.1 General Scheme

We represent the balanced sequence of parentheses as a bitmap, where 0 represents an opening parenthesis and 1 a closing parenthesis. The $rank(i)$ operation (Section 5) makes our $excess(i)$ query extremely simple, as it is the number of 0's

[4]Again, this could be alleviated somehow by choosing other constants for the terms given in the theoretical paper, yet the general conclusion would be the same. In particular, as the space is reduced via manipulation of constants, the "constant time" achieved in theory worsens in practice due to the same manipulations.

[5]In 2004, a practical implementation appeared [Geary et al. 2004] offering decent practical performance and the original theoretical guarantees. We have been unable of obtaining their code, so we recently implemented it and found that their solution is slightly smaller and slower. Our implementation is still a better choice for our application.

minus 1's. This is a simple function of $rank(i)$, namely $excess(i) = i - 2 \times rank(i)$. We use the $excess(i)$ function several times in which follows.

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

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

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

6.2 Hashing

Implementing the hashing scheme is not as trivial as it might seem. We exclude perfect hashing, whose implementation is not practical for our very large data sets.[6] We opt for a closed hashing scheme with table size at least 1.8 times the number of elements, which is a good tradeoff between space and search time (expected number of accesses per search is 2.25).[7] Hashing is done by multiplying the key by a large constant prime, and rehashing is done by adding another such prime (so we suffer from secondary clustering, but address computation is cheaper). The table size is a power of 2 to avoid computing modulus. This turned out to be much faster at a small price in space.

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

Imagine that we search for a given parenthesis position in our hash table and recover a set of possible answers (corresponding to other keys that have collided

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

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

with our key). Each such answer gives a candidate distance to our matching parenthesis. We first discard those candidates whose excess is different from that of our key. However, there could be still several candidates with the correct excess. If our key is in the table, then our solution is in the candidate set and it can only be *the closest matching position with correct excess*, since candidates that appear before the correct answer must have higher excess on a balanced sequence.

This property gives us a simple solution. We first try to find the matching parenthesis by brute force up to distance b. If we fail, then the parenthesis is either near or far. We search the hash table of near parentheses. If we find candidates with the same excess, then the closest one is the answer. If we do not, then we search the far table, where the answer surely is.

We remark that, although in general we speak of "sets" of colliding answers, in practice, we expect typically at most two colliding answers.

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

6.3 Brute Force Search

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

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

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

6.4 Solving *enclose*(i)

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

There is no complexity guarantee for this operation except the arity of the parent. In our application, this will be σ in the worst case. In practice, this

turned out to be rather slow, and the operation turned out to be needed in many places of the overall scheme, ruining the overall performance.

Hence, we preferred to directly store the enclosing parenthesis of each opening parenthesis. This is the last parenthesis before i whose excess is 1 less than the excess of i. It turns out that the hashing scheme described above fits perfectly well, and collisions can be handled the same way. The brute force search for close parentheses is also very similar to that of *findopen*(). Moreover, *findopen*() itself is not used elsewhere in our application, so we do not need to store information on closing parentheses.

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

6.5 Evaluation

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

The optimum is $s = b\log_2 p$. If $b = \log_2 p$ and $k = \log_2 p/\log_2\log_2 p$, then $s = \log_2^2 p$. Our search cost remains $O(\log\log p)$, and the space requirement is sublinear and reasonable. In our example of 64 megabits, and assuming $\sigma = 4$, we have an overhead of 2.28 times the stream size, plus a constant number of 1,248 bits (recall that this is an upper bound, it is much better on average). Using hash tables that are powers of 2 may drive this extra space up to 4.56 in the worst case.

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

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

Figure 3 shows the space overheads incurred in our example tests. On the left, we have plotted the percentage of "near" and "far" parentheses both in

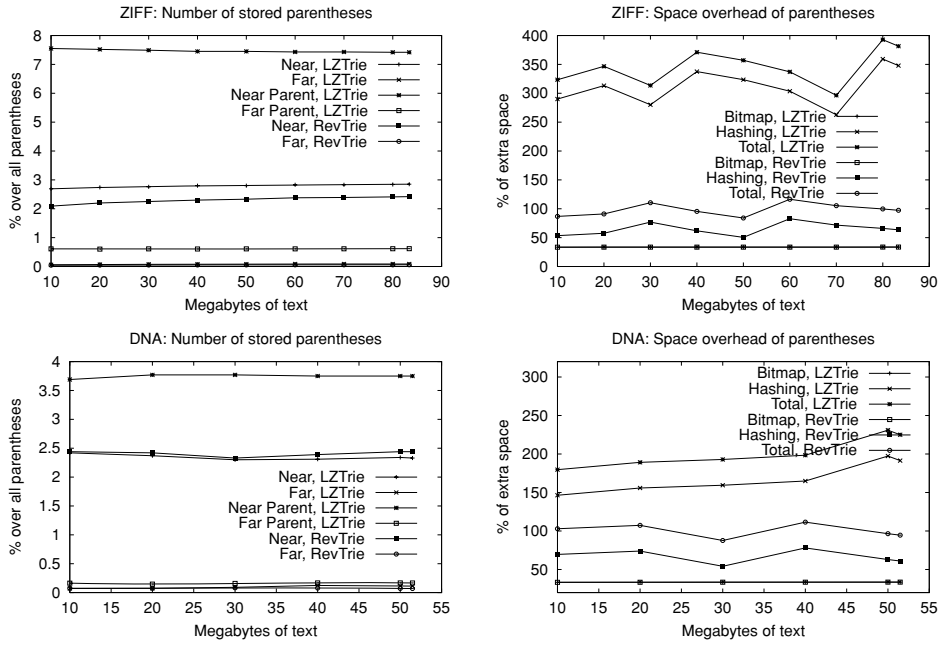

Fig. 3. Different aspects of the space overhead to store balanced parentheses.

LZTrie and *RevTrie* ($parent(i)$ is not used on *RevTrie*). On the right, we have plotted the overall space overhead to represent balanced parentheses, including those attributed to the bitmaps. As we can see, only 2% to 7% of the parentheses have to be stored in the "near" hash table, and 0.04% to 0.6% are stored in the "far" tables. The percentages seem to stabilize for large texts. The overhead due to these hash tables is, for *LZTrie* 1.5 to 3.6 times the space of the parentheses themselves, and 0.5 to 0.8 for *RevTrie*. The reason for the difference among the tries is the lack of $enclose()$ information on *RevTrie*. The reason for the fluctuations in the figures is that we require the hash table size to be a power of two to avoid computing modulus. We could change this decision, but we would pay much more in terms of search time, and these fluctuations hardly influence the overall index size (see later discussion).

7. IMPLEMENTING THE LZTRIE

We again choose to change the theoretical approach of Munro and Raman [1997], although we again reuse some of their ideas.[8]

7.1 Structure

Instead of converting our alphabet to binary and representing the trie as a binary tree and, 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

[8]A beautiful representation for general trees permitting fast branching appeared later [Benoit et al. 2005], and it has been used to implement newer versions of the LZ-index.

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

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

7.2 Operations

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

Figure 4 shows the number of invocations to *findclose*() per call to $child(i, a)$. It shows that in practice the cost of this operation in the case of DNA is around 2. For ZIFF, on the other hand, it is around 10 to 12. It is much higher, however, for short patterns (because for them most of the searches occur in the top of the trie, where arities are larger). For short patterns, the cost of the trie search is negligible compared to the overall search cost, as will be clear later.

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

The other functions we have to provide are easily implementable. First, $subtreesize(i)$, the number of nodes of the subtree rooted at i, is simply half the number of parentheses enclosed by node i, $subtreesize(i) = (findclose(i) - i + 1)/2$. Second, $depth(i)$, the depth in the tree of node i, is exactly $excess(i)$. Third, $leftrank(i)$ and $rightrank(i)$, the first and last lexicographical positions of strings below node i, are simply $leftrank(i) = i - rank(i)$ (as node i represents the smallest string in the subtree) and $rightrank(i) = j - rank(j) - 1$, where $j = findclose(i)$. Fourth, $rth(pos)$, the block identifier of the pos-th string in the trie, is just $ids(pos)$. Fifth, $ancestor(i, j)$[9] tells whether node i is an ancestor of j, and is just $ancestor(i, j) = i \leq j \leq findclose(i)$. Finally, the letter and identifier of node i can be easily rewritten as $letter(i) = lets(leftrank(i))$ and $id(i) = ids(leftrank(i))$. We also implement simple functions that permit traversing the trie in depth first search order (not caring about the letters), which is useful to report whole subtrees (occurrences of type 1).

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

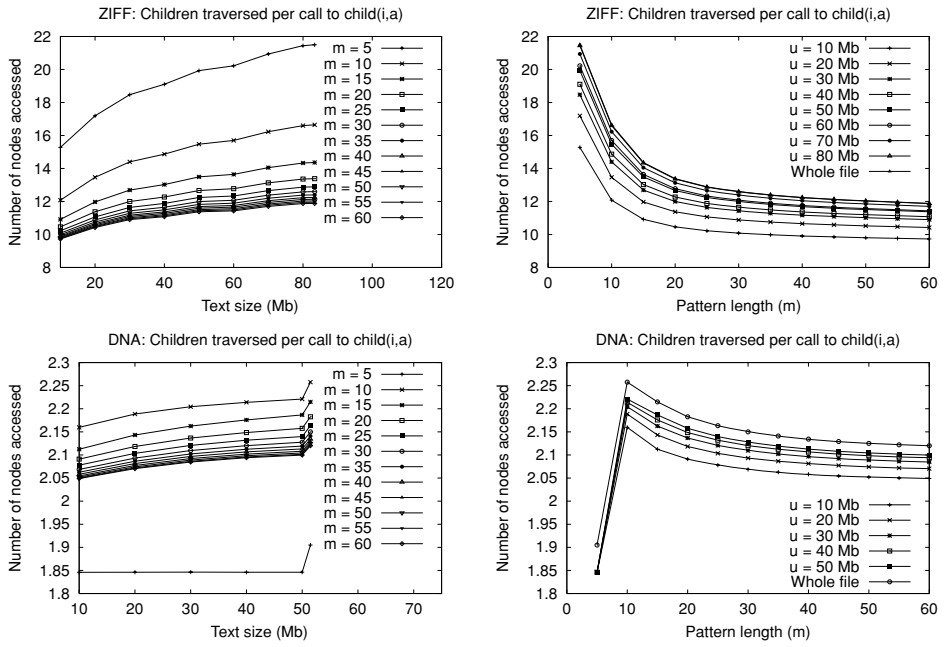

Fig. 4. Number of *findclose*() invocations to find the proper child in the $child(i, a)$ operation.

7.3 Construction

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

Figure 5 (left) shows the construction times for *LZTrie*. We have identified four steps: (1) creation of normal trie, where the text is read and the normal trie (with pointers) is built; (2) representation of this trie using parentheses, which involves simply traversing it and writing down bits when a new tree is started/finished; (3) freeing the normal trie, which is essential to limit the overall construction RAM space; and (4) creation of the compressed trie representation, which means creating a balanced parentheses data structure using the bit stream that represents the trie and creating the arrays of letters and identifiers.

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

In fact, freeing the trie was initially time-consuming too, as we had to free all the individual nodes. As there were few different sizes to work with, we

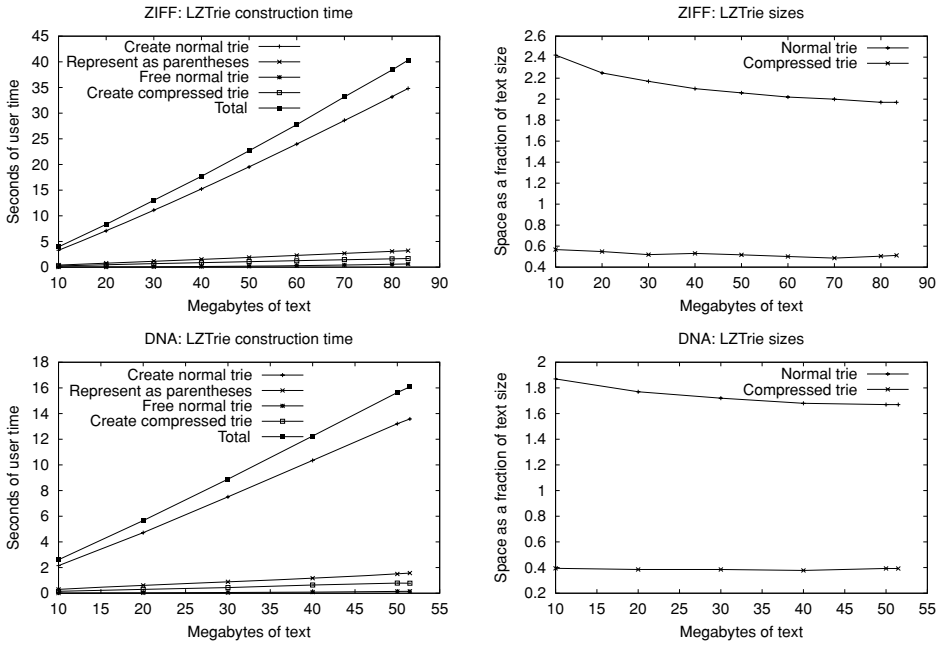

Fig. 5. Different aspects of the construction of the *LZTrie* data structure.

decided to create our own memory manager handling objects of fixed size, using a different manager for each different object size. This manager allocates nodes in blocks, and freeing them is much faster than the naive approach. Some programming languages (like Modula-2) or runtime support systems provide independent "heaps", that is, memory areas that are allocated and freed once and that support allocation of memory inside them. This kind of heap is what we are simulating.

7.4 Extra Space

Figure 5 (right) shows the extra space needed by *LZTrie*. The larger value is the space (as a fraction of the text size) needed by the normal trie. The smaller value is the space after we have it in compressed form. It can be seen that the space requirement drops as the text grows, which is a consequence of having a number of nodes equal to the number of blocks, which grows as $n = O(u/\log u)$. For large texts, the extra space needed to build the normal trie becomes 1.7 to 2.0 times the text size. This space is freed as soon as we have enough information to build the compressed representation.

8. IMPLEMENTING REVTRIE

8.1 Structure

The reverse trie has several similarities with *LZTrie*, but also some important differences. The trie is also represented by a sequence of balanced parentheses and a sequence of block identifiers (*rids*), but this time (1) the edge between two nodes can be labeled by a string, which is not represented; (2) we remove

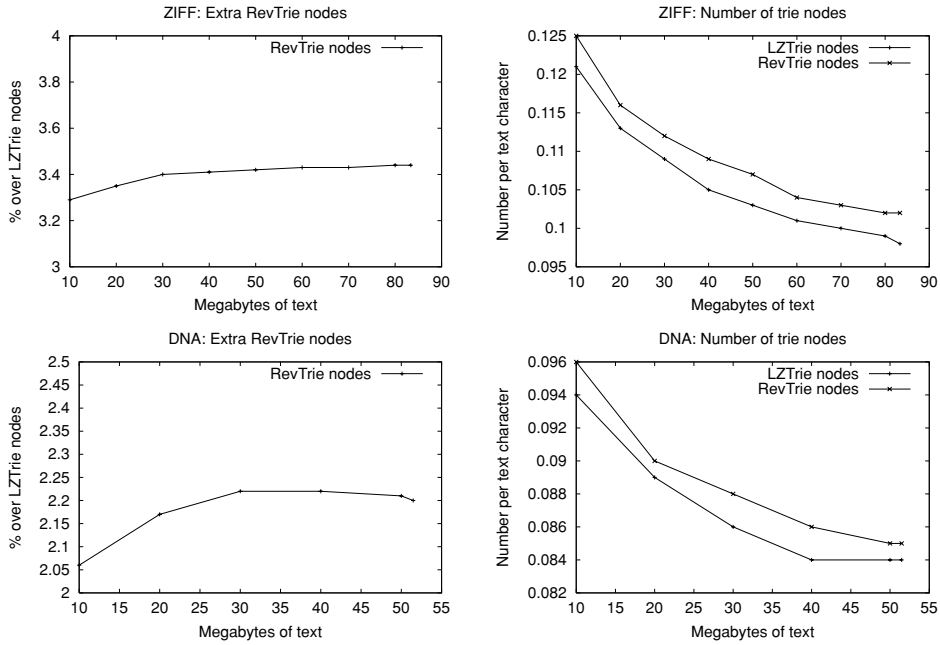

Fig. 6. Number of trie nodes in *LZTrie* and *RevTrie* data structures.

unary nodes that have no block identifier, but still nonunary nodes without block identifiers remain and are represented (these will be called *empty* nodes); and (3) we do not implement the *parent* operation.

A first question is which is the number of nodes of *RevTrie*, since we have the same n nodes of *LZTrie* plus some empty nodes. As shown in Figure 6 (left), the extra *RevTrie* nodes stabilize around 3.4% over *LZTrie* nodes on ZIFF and around 2.2% on DNA. So the price for these empty nodes, which are extremely convenient for search purposes, is minimal. On the right, we can see that the number of trie nodes per text character decreases, as expected from a Ziv-Lempel parsing. What may be not so clearly expected is that *RevTrie* nodes decrease at the same rate, posing a constant overhead over *LZTrie* nodes.

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

8.2 Operations

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

If and when we determine that j is the correct child of i by letter a, we have to determine which is the string that joins i to j. This string can be obtained by going up more steps in *LZTrie*, but we need to know where to stop. Since, by construction of *RevTrie*, the identifier of j is that of the lexicographically smallest string in the subtree (be j an empty node or not), we only have to compute $rightrank(j)$ to have the smallest and largest strings in the subtree. We map also $j'_t = Node(rth_t(rightrank(j)))$ to *LZTrie*, and go up string xa. At this point, we go up from both nodes in *LZTrie*. As soon as their characters differ, we have read the string that joins i to j. This process is done interactively with the calling method to solve the next calls to $child(child(i, a), b)$ fast. Only when we finally arrive at j, we have to rescan the set of children, go up their paths in *LZTrie*, and so on. Nevertheless, the process is tedious and slow, so we seek to limit it as much as possible.

8.3 Construction

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

Figure 7 (left) shows the construction times for *RevTrie*. We have identified the same four steps as for *LZTrie*. Again, the most time-consuming process is by far the creation of the trie. Although a bit better than the construction costs for *LZTrie*, these times dominate the overall construction cost. Again, we can see a slightly superlinear increase due to caching.

8.4 Extra Space

What has worsened a lot is the extra space needed to hold the normal trie. Figure 7 (right) shows that we need around four times the text size for ZIFF, and 2.5 times for DNA. This shows that the extra space to represent unary empty nodes is around 77% to 85% for ZIFF and 35% to 50% for DNA. After we compress unary paths, the compressed representation becomes even smaller than that of *LZTrie*, as we do not store the *lets* array. The space of the normal trie is freed as soon as we have enough information to build its compressed representation, but it influences the maximum amount of main memory we need across all the indexing process. In particular, using path compression at the time of the construction of the normal trie would greatly reduce the overall space requirement.

9. IMPLEMENTING NODE AND RNODE INSTEAD OF RANGE

Node is just a bit stream with as many bits as necessary to represent n nodes of *LZTrie* (i.e., positions 0 to $2n - 1$). It is built as the inverse of the array *ids* of *LZTrie*. What is more interesting is the implementation of *Range*, or more appropriately, its replacement by a *Node*-like data structure.

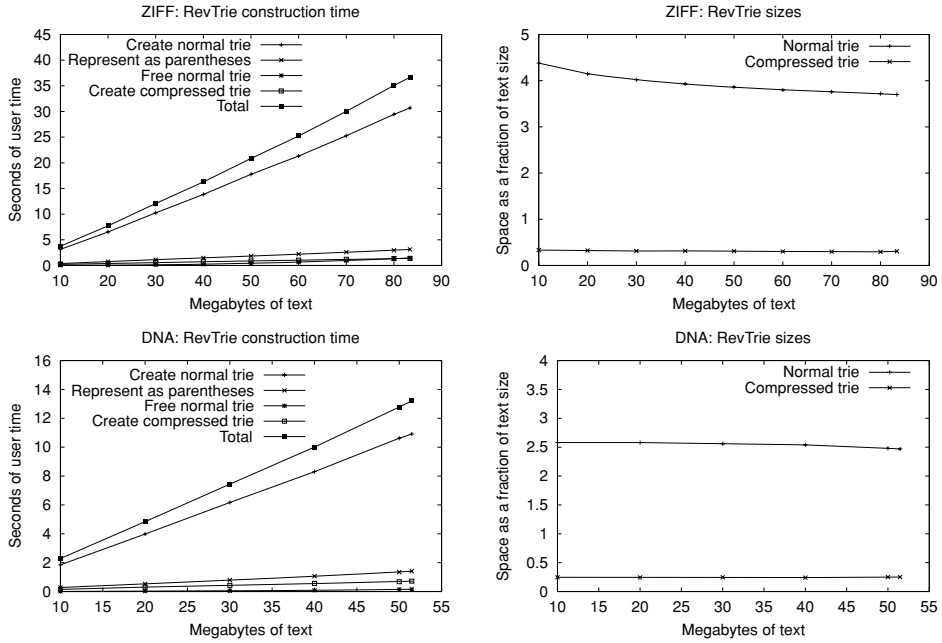

Fig. 7. Different aspects of the construction of the *RevTrie* data structure.

In the beginning, we implemented the *Range* data structure pretty much as described by Chazelle [1988] (Section 4.1). The only interesting improvement was that we noticed that, if the points (x, y) represented a bijection, then it would be possible to know the initial position of the right subtree (in the first level there would be exactly $2^{\lceil \log_2 n \rceil}/2$ zeros, etc.). This would save us some *rank*() computations and would permit an easy construction algorithm. Hence, we completed the mapping with fake values so as to make it a bijection (originally it mapped *LZTrie* node labeled $k+1$ to *RevTrie* node labeled k, so it was not a bijection because there are extra empty *RevTrie* nodes). The price in space was rather moderate and the structure was faster.

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

With *RNode*, we could solve quite decently the same problem addressed by *Range*, as follows. Say that the search for $P^r_{1\ldots i}$ in *RevTrie* leads us to node i_r, and the search for $P_{i+1\ldots m}$ in *LZTrie* leads us to node i_l, If any of the two nodes does not exist, we know immediately that this partition of P produces

no matches.[10] Both for i_t and i_r, we can use *rank* and *rightrank* to determine the ranges in the *ids* arrays where the relevant blocks lie. Then we have two choices:

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

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

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

However, as we will see later, the cost of this part of the search becomes much less important for that length, as other parts dominate. In practice, we found that the version based on *RNode* took half the time of *Range* on short patterns (for example searching for all the English words of a dictionary on ZIFF) and two-thirds the time on long patterns, no matter the length of the text.

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

10. INDEXING AND ITS PERFORMANCE

Construction of the overall index proceeds as follows:

1. We build the normal Ziv-Lempel trie.
2. We represent it with parentheses, letters and identifiers. The array of identifiers is not yet bit-packed but represented as computer integers.

[10]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.

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

This sequence is designed to minimize the maximum amount of main memory required to index. The maximum is usually reached after step 10. Note that we soon free the text and never need it again. Figure 8 (left) shows the maximum amount of main memory required at indexing time. As shown, the percentage of extra space required drops as the text grows, influenced by the smaller amount of trie nodes. For ZIFF, we need from 4.8 to 5.8 times the text size, whereas for DNA, this drops to 3.4 to 3.7. As a comparison, the construction of a plain suffix array without any extra data structure requires five times the text size. The difference, of course, is that, after we build the index, we are left with a very succinct representation, while a normal suffix array needs those five times the text size forever.

Figure 8 (right) shows construction costs. It becomes clear that the construction of the tries dominates the overall construction cost, and that this is slightly superlinear. In the case of ZIFF, for 10MB, we build the index at 0.82 sec/MB, whereas for the whole 83.37MB text, this has grown to 0.97 sec/MB. For DNA, the figures are 0.53 sec/MB on 10MB and 0.61 sec/MB on 51.48MB. In general, we can speak of a construction speed of 1 to 2MB/sec, which is much better than the construction costs of, for example, suffix arrays.

Finally, Figure 9 (left) shows the space requirement of the finished indexes. The oscillations of the parentheses representation are hardly noticeably. The space requirement drops as the text sizes grow. For ZIFF, we require from 1.6 to 1.4 times the text size (and this includes what we need to reproduce the text), whereas for DNA, the figure stays around 1.2.

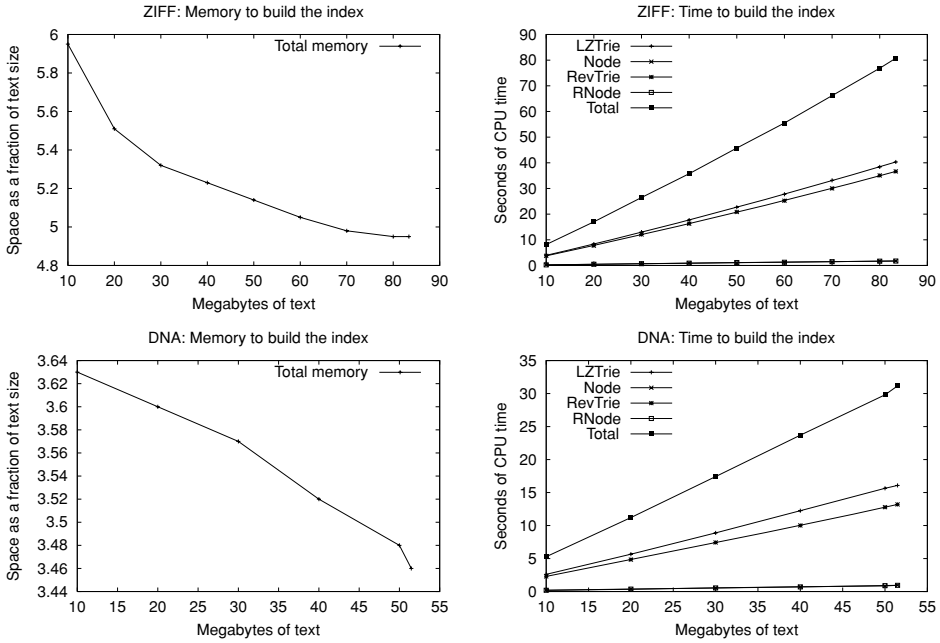

Fig. 8. Different aspects of the construction of the whole index.

The compression ratios obtained are related to the compressibility of the text. To show this more clearly and to test our predictions of using four times the space needed by the compressed text, we show the space of the final index as a fraction of the file size after we compress it with Unix's *Compress* program (an LZW compressor), as well as as a fraction of $n \log n$, being n the number of LZ78 blocks. The latter accounts for a pure LZ78 compression scheme and should give a better estimate of the Ziv-Lempel compressibility of the text. Compress, on the other hand, gives a real-life estimate of the achievable compression and is a useful control value.

As shown in Figure 9 (right), Compress obtains compression ratios which are about 20% over the theoretical LZ78 optimum (LZW needs just $\lceil \log n \rceil$ bits per phrase). Our indexes take 3.6 to 4.7 the space achieved by Compress. If we compare against the theoretical optimum, we have that our index takes about five times the size of the Ziv-Lempel compressed text. Our prediction was four times that size; the rest accounts for terms considered sublinear: the arrays of letters and of parentheses, hash tables, etc.

11. SEARCHING WITH THE INDEX

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

To analyze the empirical behavior of the search process, let us call ℓ the average length of a LZ78 phrase (that is, $\ell = u/n$ if u is measured in bytes and n in blocks). It holds $\ell = \Omega(\log u)$. The larger ℓ, the more compressible the

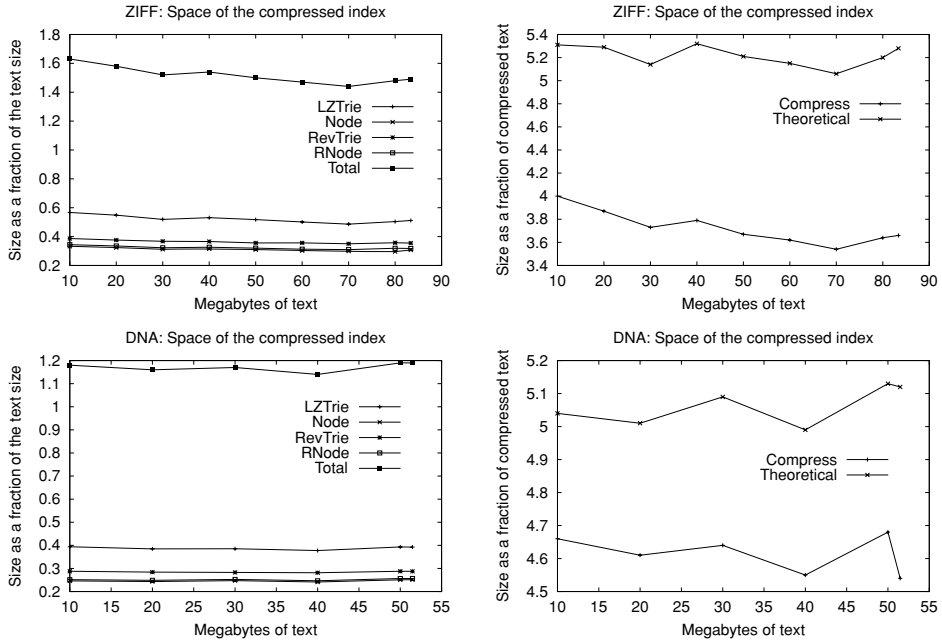

Fig. 9. Sizes of our compressed indexes.

text is, as it can be represented in $(u/\ell)\lceil \log_2 u/\ell \rceil$ bytes. Except for very special texts, most phrases have length ℓ, and also $\ell = \Theta(\log u)$. In our experiments (Figure 6), we obtain good approximations of the form $\ell = 6.16 + 0.22 \ln u$ on ZIFF and $\ell = 8.74 + 0.18 \ln u$ on DNA.

11.1 Building Matrix $C_{i,j}$

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

On average, we do not expect to fill all the $m^2/2$ cells. Only strings up to length ℓ appear in *LZTrie*, and hence we expect to fill $O(m\ell)$ cells of $C_{i,j}$. This is confirmed in Figure 10, where we have drawn the number of cells really filled and an $O(m \log u)$ pattern is rather clear.

Filling each cell involves a $child(i, a)$ operation, whose cost is proportional to the average arity of the trie. Most of these operations are done close to the root, where the arity is close to its maximum σ. Hence, on average, we work $O(\sigma m \min(m, \log u))$ time to fill $C_{i,j}$. This is obviously a simplification if we regard Figure 4. However, it works well because, for large m, the average arity traversed stabilizes (around 11 for ZIFF and 2 for DNA). For small m, on the other

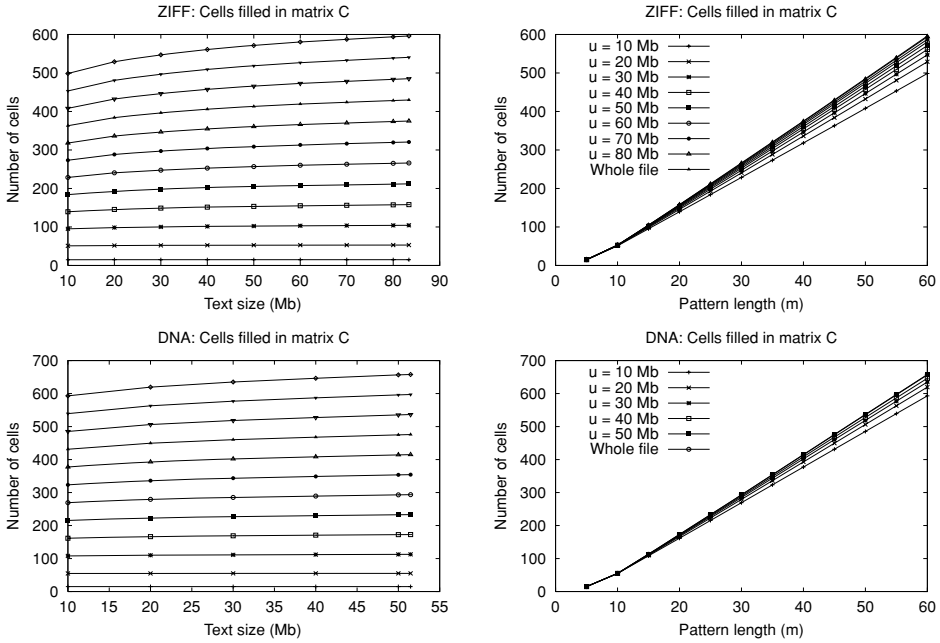

Fig. 10. Number of cells actually filled in matrix $C_{i,j}$. On the left, the lines represent, from lower to upper, $m = 5$, $m = 10$, and so on until $m = 60$.

hand, the times are negligible. The difference in arity (11 to 2) is explained if we define σ as the inverse of the probability of two random characters being equal for each text. This gives $\sigma = 4.8$ on DNA (recall that there are newlines and N's) and $\sigma = 20.0$ on ZIFF.

Let us now focus on real experimental times. Figure 11 shows the time to fill matrix $C_{i,j}$ (and $Cid_{i,j}$). The expected $O(m \log u)$ pattern is still visible, although a bit hidden by variance. We have used least squares with the model $t = a + b\sigma m \ln u$ and obtained

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

with a percentual error[11] below 7% in both cases. (By μsecs we mean microseconds.)

11.2 Building Vector B_j

The second step searches for every reversed pattern prefix, $P^r_{1...j}$, in *RevTrie*, and stores it in an array B_j. This is necessary to report occurrences of type 1 and 2. Since searching in *RevTrie* is much slower than on *LZTrie*, we seek to reduce this work as much as possible. The results already obtained in Cid are useful. If we look for $P^r_{1...j}$ and $P_{1...j}$ exists in *LZTrie* (i.e., $C_{1,j}$ is not null), then $RNode(Cid_{1,j})$ directly gives us the corresponding node in *RevTrie*. Otherwise,

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

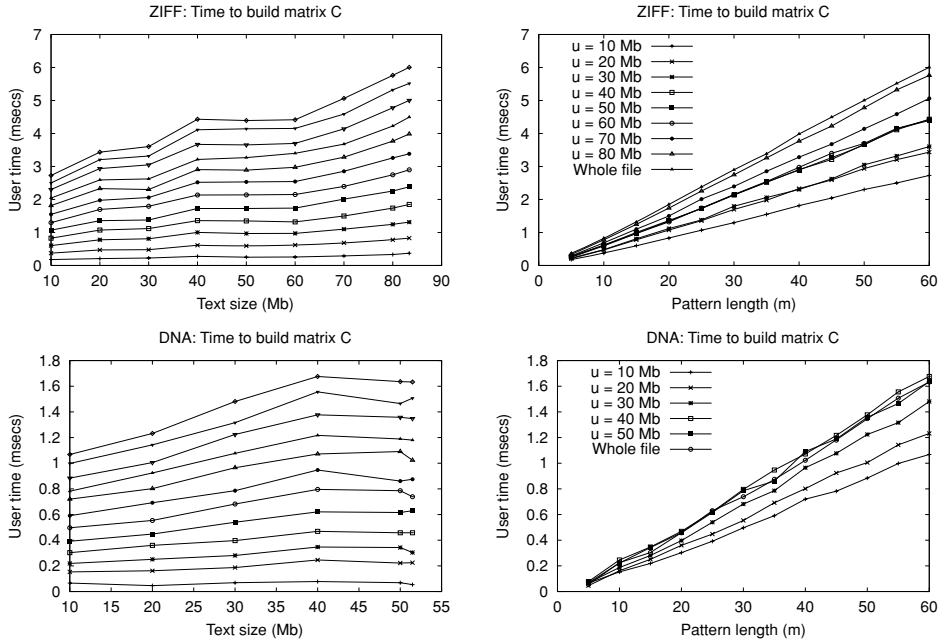

Fig. 11. Time to build matrix $C_{i,j}$. On the left, the lines represent, from lower to upper, $m = 5$, $m = 10$, and so on until $m = 60$.

$P^r_{1...j}$ corresponds to an empty node or to a position in a string between two nodes and cannot be directly found with *LZTrie*. Still, we can reduce the search cost as follows. Let i be the minimum value such that $C_{i,j}$ is defined. Then, $RNode(Cid_{i,j})$ is the lowest nonempty ancestor of the node we are looking for. We reduce the work to that of searching for $P^r_{1...i-1}$ starting from node $RNode(Cid_{i,j})$. This final partial search must be done using repeatedly operation $child_r(node, a)$ (once per node arrived at).

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

Figure 12 shows that, to build vector B_j, we are working over $O(m)$ cells. Indeed, we work over a 10% of the cells filled to build matrix $C_{i,j}$.

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

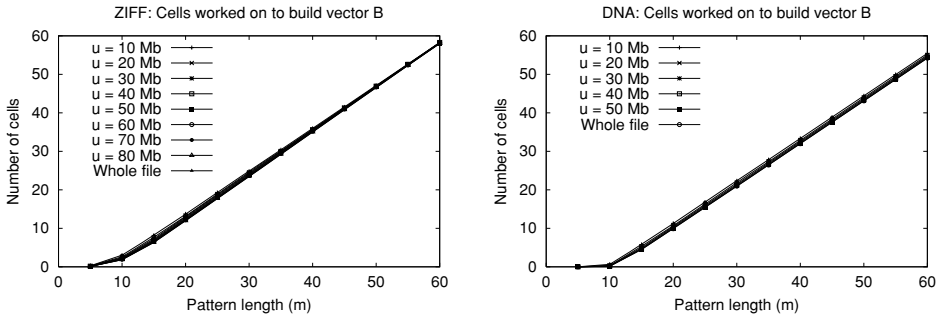

Fig. 12. Cells worked on to build vector B_j.

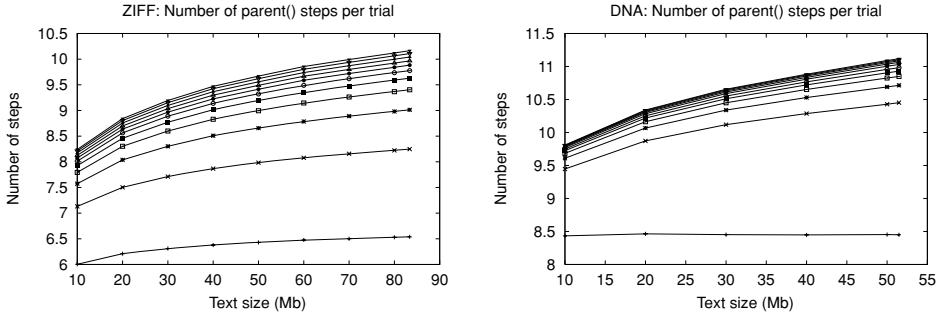

Fig. 13. Number of *parent*() operations per *child*() trial. The lines represent, from lower to upper, $m = 10$, $m = 15$, and so on until $m = 60$.

Figure 13 shows the number of *parent*() operations performed per *child*() trial. The case $m = 5$ is excluded from the figures because no cells of B_j were worked on in that case, as all could be predicted using $C_{i,j}$ (case $m < \log u$). The logarithmic pattern is clear.

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

Figure 14 shows the time to fill vector B_j. As shown, it is much smaller than the time to fill $C_{i,j}$ (at least 3 times smaller on DNA and 10 on ZIFF), despite that in principle filling B_j is much more complex. This is an achievement of our techniques to reduce the amount of computation as much as possible. It can also be seen that the times are essentially linear in m and depend slightly on u.

Using least squares with the model $t = a + bm + cm \ln u$ (as the logarithmic pattern of Figure 13 is of the form $x + y \ln u$) we obtain

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

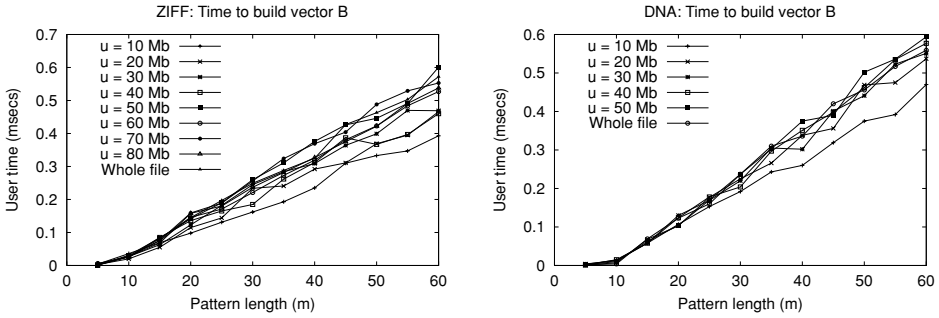

Fig. 14. Time to build vector B_j.

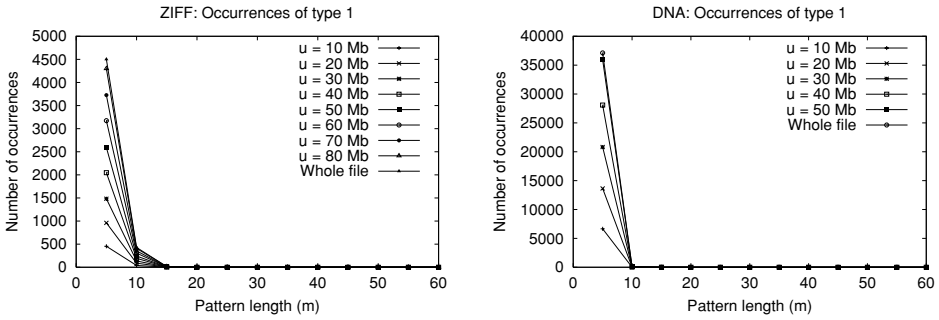

Fig. 15. Number of occurrences of type 1.

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

11.3 Reporting Occurrences of Type 1

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

Figure 15 shows the number of occurrences of type 1. As shown, these are significant only for short patterns. The probability of an occurrence being of type 1 is that of a block beginning not falling at positions 2 to m, that is, $(\ell - m + 1)/\ell$. This is confirmed in Figure 23, where the ratios for $m = 5$ are 0.66 on DNA (predicted 0.66) and 0.58 on ZIFF (predicted 0.60).

Figure 16 shows the times. As shown, these are significant only for very frequent patterns ($m = 5$), as the other ones almost have no occurrences of this type. It is also clear that the time to report the occurrences is almost 15 times that of just counting them (in part because we can count whole subtries in one shot). We count about 15,000 occurrences per msec, and report about 1,200 per msec.

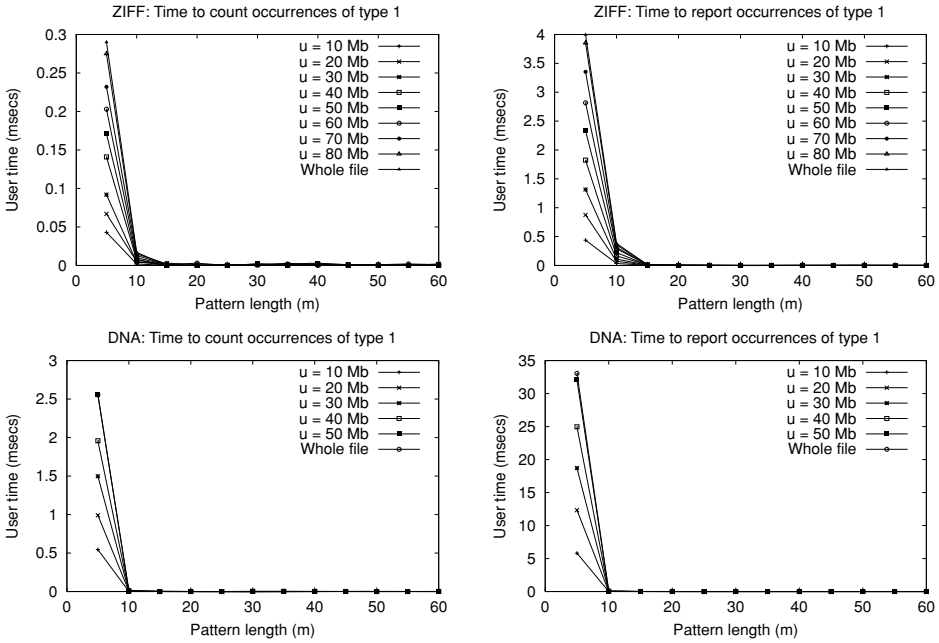

Fig. 16. Time to count (left) and to report (right) occurrences of type 1.

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

$$count_1 = 0.067\,(\ell - m + 1)/\ell\; R \quad \mu secs$$
$$report_1 = 0.83\,(\ell - m + 1)/\ell\; R \quad \mu secs$$

11.4 Reporting Occurrences of Type 2

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

Figure 17 (left) shows the number of occurrences of type 2. On the right, we show the amount of verification work, that is, number of candidate nodes tested. On ZIFF, 1 out of 3.5 becomes a real occurrence, while on DNA only 1 out of 9.5.

Figure 18 shows the times. Again, they are significant only for frequent patterns ($m \leq 15$), as the other ones almost have no occurrences of this type in the text. This time, however, longer patterns have significant times.

The real time for this step depends both on the number of candidates and of real occurrences. We test about 3,500 candidates per msec, considering counting time. Reporting these real occurrences is done at a rate of 1,400 per msec. There

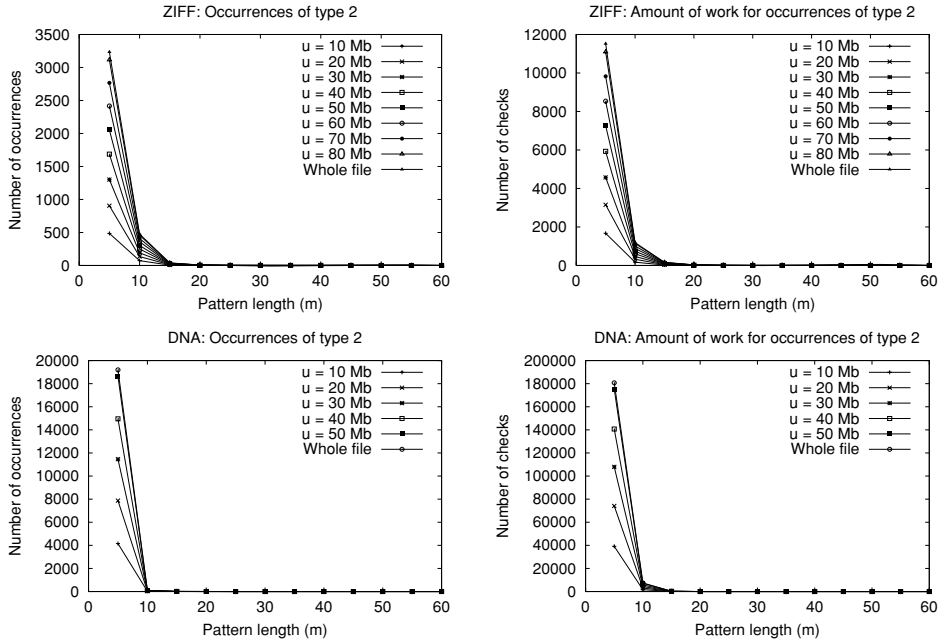

Fig. 17. On the left, number of occurrences of type 2. On the right, amount of verification.

is extra work to test all the splitting positions, but this turns out to be absolutely negligible. Occurrences are reported a bit faster than those of type 1 because, in this case, we have direct access to the nodes, without the need of traversing the trie. However, the difference is not too significant.

The main issue is to predict how much verification work will we perform. The most important length we have to care about is $m/2$, as the number of verifications triggered by the longer ones is much smaller. We can expect to find $R = u/\sigma^m$ occurrences, while the number of candidates is on average $u/\sigma^{m/2}$. From these, however, only 1 out of ℓ is placed at a text position such that there is a block boundary at position $m/2$. Hence, we expect only $u/(\ell\sigma^{m/2})$ candidates, which is $\sqrt{uR}/\ell$. This estimation turns out to work more or less well for DNA, where it predicts 80% of the real amount of verifications, but very bad on ZIFF, where it predicts 700% of the actual number.

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

Let us call σ_k the inverse probability of matching if we consider k previous characters (hence $\sigma = \sigma_0$). We expect that 1 out of $\sigma_3\sigma_4$ candidate occurrences become real occurrences for $m = 5$. That is, the fourth character has to match given the first 3, and then the fifth has to match given the first 4. This gives us

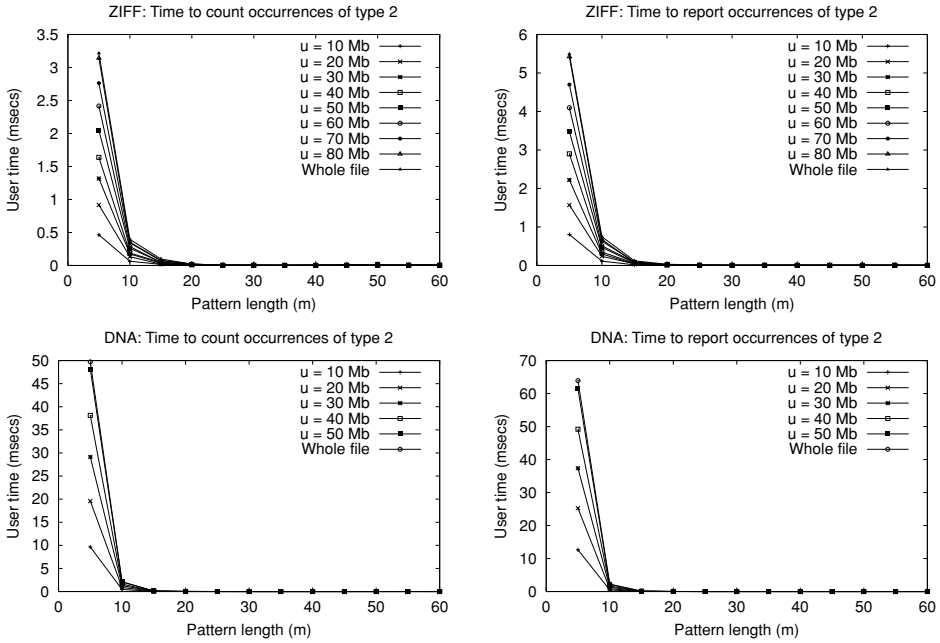

Fig. 18. Time to count (left) and to report (right) occurrences of type 2.

Table I. Inverse of the Probability of two Characters Matching, Given Contexts of Different Orders

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

1/4.48 on ZIFF and 1/8.82 on DNA. This is a much better estimation, especially because, for memory limitations, we could not compute exactly the value σ_4 on ZIFF. We expect, therefore, at counting time, a verification effort over $R_2W = R_2\sigma_{\lceil m/2 \rceil} \ldots \sigma_{m-1}$ candidates, where R_2 is the number of occurrences of type 2. On a random, text $\sigma_k = \sigma$ and hence $RW = R\sigma^{m/2} = u/(\ell\sigma^{m/2}) = \sqrt{uR}/\ell$ as explained. We can thus predict

$$count_2 = 0.29\,(m-1)/\ell\; RW \quad \mu\text{secs}$$
$$report_2 = 0.71\,(m-1)/\ell\; R \quad \mu\text{secs}$$

11.5 Reporting Occurrences of Type 3

Instead of the arrays A proposed in the theoretical part, we opt for a closed hash table (load factor at most 1/2) where all the triples $(i, j, Cid_{i,j})$ are stored with key $(i, Cid_{i,j})$ (of course only when Cid is not null). Then, we try to extend each match $C_{i,j}$ by looking for $(j+1, j', Cid_{i,j}+1)$ in the hash table, marking

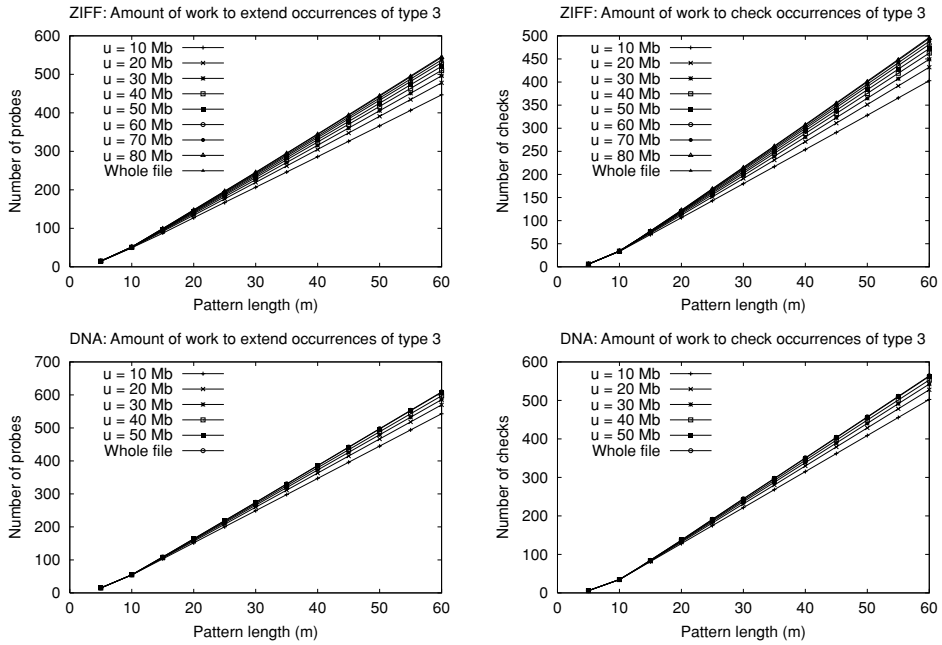

Fig. 19. On the left, number of probes to extend maximal occurrences of type 3. On the right, number of checks of maximal occurrences.

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

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

Figure 19 shows the amount of probes to extend maximal occurrences (left) and number of maximal occurrences found and checked (right). As expected, both are very similar, and they increase linearly with m and slightly with u.

Figure 20 shows the times. They are rather negligible overall, although they increase linearly with m and slightly with u. The number of occurrences found is very low, so the time is basically that to extend occurrences and check maximal ones. Our estimation is:

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

with percentual errors below 20% to 25%.

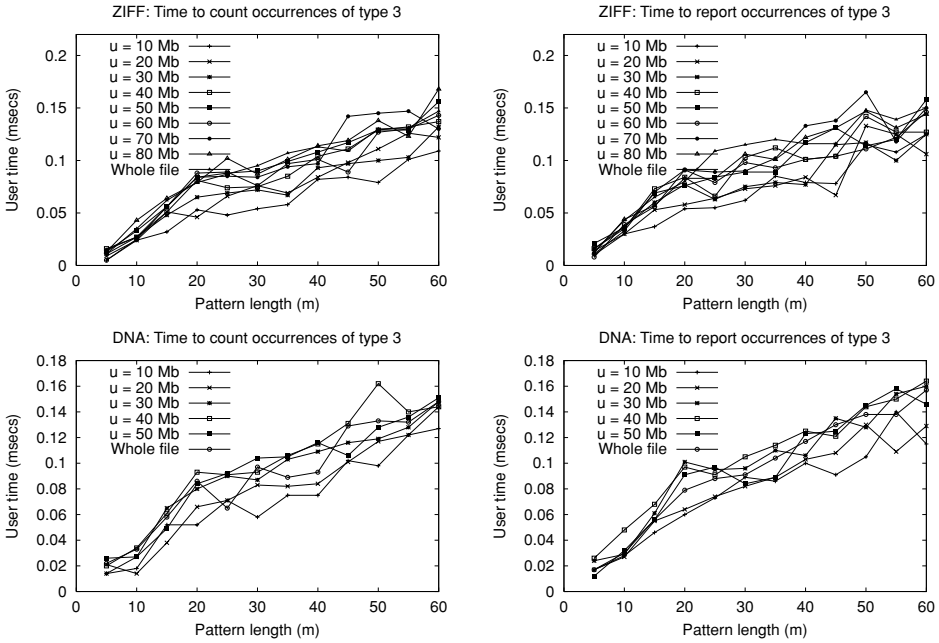

Fig. 20. Time to count (left) and to report (right) occurrences of type 3.

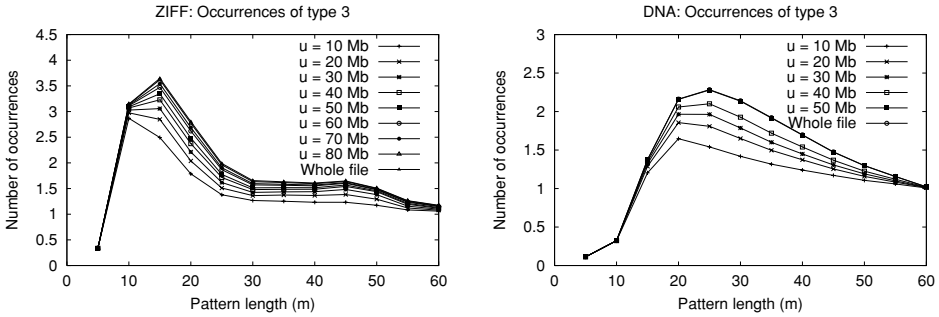

Fig. 21. Number of occurrences of type 3.

It is somewhat interesting to see the number of occurrences of type 3, shown in Figure 21. It first increases (because short occurrences cannot span three blocks) and then decreases as occurrences are less probable. At the end, it converges to one because patterns are taken from the text. In all cases, they are very few.

11.6 Reporting Levels

Reporting is done at three levels. The lowest level is counting: We just tell how many times P occurs in T. The next level (which we have called just "reporting") gives all the text positions of P in T. These are given in the form (block number, offset). Converting them to usual text positions would require another

array that maps block identifiers to text positions. This can be done,[12] but we consider that it is usually not necessary. The representation we use can be used to compare two positions so as to determine which is smaller, and to obtain the surrounding text given a position in that format. This is enough for most applications, although not sufficient for those that need to, for example, determine the text distance between two occurrences. Finally, the highest reporting level prints the text around each occurrence found. Currently, the context is limited by newlines, but this is not hard to change. This text is obtained backwards by moving from the block of interest towards the root of *LZTrie* and printing the letters found at the edges. More and more blocks to the left or to the right are easily obtained using $Node(k-1)$ or $Node(k+1)$.

Figure 22 shows the overall query times under the different reporting levels. Note that we use a logarithmic scale on y. The nonmonotonic behavior comes from the fact that, for short patterns, reporting time largely dominates, while for long patterns the most relevant time is that of searching for the pattern substrings.

Figure 23 shows the overall number of occurrences. This shows that we report 600 to 800 occurrences per msec, and output about 14 matching lines per msec.

12. COMPARISON AGAINST OTHERS

We have compared our prototype against two of the most prominent alternative proposals.[13] We have used the whole ZIFF and DNA texts. We have considered construction time and space, but our highest interest is in query times, both for counting and for reporting.

An important fact is that the different indexes take different space. Although our index does not permit important space-time tradeoffs,[14] the others do. Hence, we tune the other indexes so they take the same space as ours.

All these indexes need to operate the data structure in main memory, as their access pattern to the structures is random. Hence, it should be made clear what we mean by "the space of the index," as we can usually store them taking less space than the one needed to operate in main memory. In one extreme, we could just map all our main memory data structures to disk and use the same disk space as the main memory space, and "booting" the index would consist of directly loading all the disk space into main memory. On the other extreme, we could simply store the plain text compressed in the best way, totally independent of our index, and "booting" would consist of actually constructing the index. As the server will probably load the index once and then answer many queries, both choices may be acceptable. In particular, depending on the index construction time versus disk speed, one or the other extreme may be the best choice. This shows that speaking of the space needed by the index on disk may have little sense, as all could just store the compressed text and build the

[12]As mentioned, it was actually done when integrating this index into *Pizza&Chili* in order to comply with its interface specification. This raised the space and reporting time only slightly.
[13]As of 2003. As explained, the conclusions roughly stay the same nowadays.
[14]Some could be achieved by enlarging hash tables, e.g., but the result is not much significant.

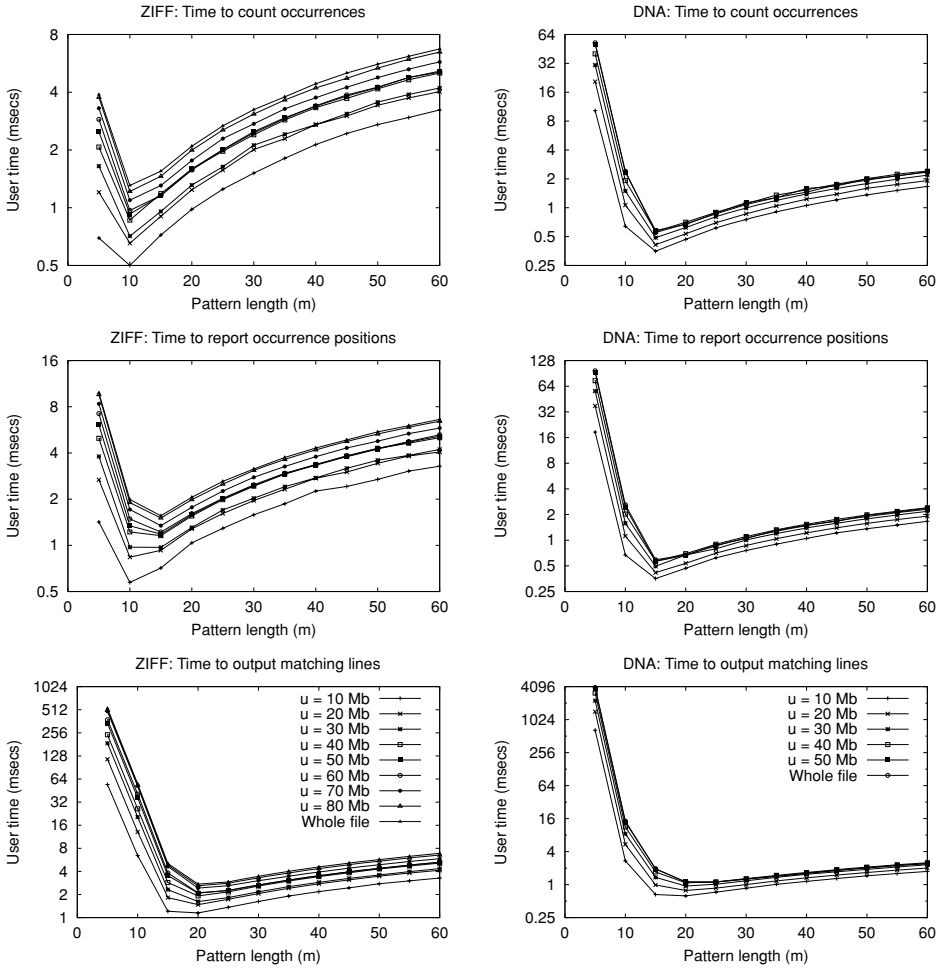

Fig. 22. Overall query times when counting occurrences (top), reporting positions (middle), and to output matching lines (bottom). The legends are on the lowest figures of each column.

index on the fly. Rather, we will be interested in how much main memory space does the index need in order to operate properly.

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

12.1 A Prototype of our LZ-index

A prototype of our index is available at `http://www.dcc.uchile.cl/gnavarro/software`. We also give some extra details that are not mentioned in our earlier description.

It indexes the text and writes down two files, storing *LZTrie* and *RevTrie*. We store only the plain sequence of parentheses, not the extra data structures to permit navigating them. We do not store *Node* and *RNode*, as these are just the inverse functions of the sequence of block identifiers *ids* and *rids*, which are

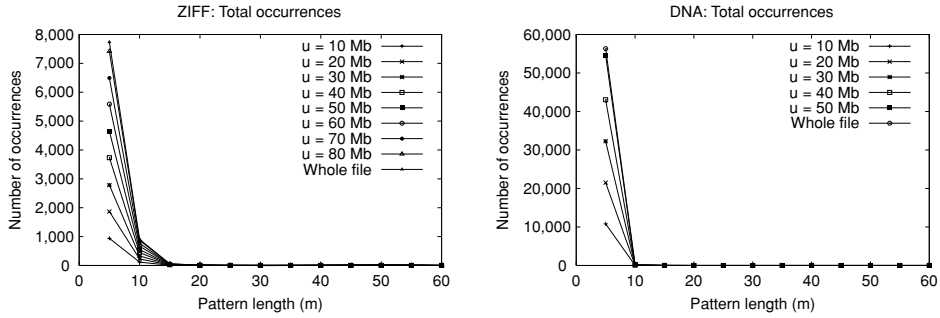

Fig. 23. Overall number of occurrences.

saved as such. The indexes on disk take 75% of the text size for ZIFF and 60% on DNA (recall that we could delete the text after the index is built).

At search time, the index is read from disk and the missing structures are computed on the fly. *Node* and *RNode* are built using no extra space apart from that of the final index in main memory. The load time is around 55 to 70 msec/MB.

12.2 Ferragina and Manzini's FM-index

This index [Ferragina and Manzini 2000, 2001] consists of (1) the permuted text, whose characters are reordered according to the Burrows-Wheeler transform and then compressed using run-length and move-to-front, (2) a two-level directory C that permits knowing $count(c, i)$, the number of times character c appears before position i in the permuted text, and (3) a sampling of pointers S that tells, for 1 out of D permuted text positions, which is their position in the original text.

There is an executable of this index at the Web page of the authors, `http://roquefort.di.unipi.it/~ferrax/Libraries/fmindex`, but the interface does not permit running massive and trustable tests, as it can search for one pattern per run.[15] On the other hand, the original design of this index aims at a data structure smaller than the LZ-index, and we could not force the executables to use all the space taken by the LZ-index (see later discussion). Hence, we implemented the index ourselves so as to take full advantage of the available space. We explain now the decisions taken. These follow rather closely the descriptions of Ferragina and Manzini [2001]. We did our best to implement this index as efficiently as possible. Later, we will give some control measures to show that our implementation is competitive against the executables given by the authors.[16]

[15]It is possible to search for many patterns in one run, but in this case only counting queries can be posed, that is, they do not report the occurrence positions.

[16]In 2005, the authors launched a second version of their FM-index, which is even more compact (and slower) than the first. This does not affect our experiments, where we aim at a larger and faster implementation instead. Both versions of the FM-index (version 2 with sources) are now available on the *Pizza&Chili* site.

Directory C works as follows. The permuted text is divided into blocks of 2^8 characters and superblocks of 2^{16} characters. For each superblock and character c, we store a cumulative count of the occurrences of c up to the beginning of the superblock. Inside each superblock, for each character c, we store a byte sequence (of length at most 2^8) of how many times does c appear inside each block of that superblock. Runs of zeros are frequent and these are run-length compressed. Hence, to compute $count(c, i)$, we have to (1) get the superblock counter value; (2) run through the byte sequence of c inside the superblock, adding per-block occurrences until reaching the relevant block (runs of zeros are traversed fast); and (3) traverse the permuted text inside the block to add up the final intrablock value. This structure permits computing $count(c, i)$ by doing simple operations over at most $2^8 + 2^8$ bytes (considering byte sequences and permuted text). Its extra space requirement is rather modest, as seen soon.

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

The idea is that, starting at some permuted text position, we move backwards position-wise in the *original* text positions, which leads us to another (basically random) permuted text position. We keep doing that until we find a position that is sampled in S, and then we know where we are in the original text and where we were when we started the process. S regularly samples the original text, so we need, in addition to the array of pointers, a hash table that answers whether a given permuted text position is sampled or not. This was implemented as a closed hash table with load factor 0.5. The idea suggested by Ferragina and Manzini [2001], of marking characters and sample their positions, was discarded because it does not permit obtaning the desired extra space on DNA.

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

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

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

The percentages given to D are so low because we decided to store the permuted text in uncompressed form. The other choice would have been keeping it compressed and using the extra space for S. The compressed texts take 23.24% of the original text on ZIFF and 25.27% on DNA (we use Huffman to compress the move-to-front values, as this gave better results than δ-coding [Bell et al. 1990]). This would have permitted us to use, for S, 2.87 times more space on ZIFF and 5.40 on DNA. This means that D would decrease 2.87 or 5.40 times. Multiplying this by the number of characters processed per block, we have that we would have processed 12.35 times fewer bytes on ZIFF and 21.37 on DNA.

The drawback, of course, is that those bytes we have to traverse now are much more expensive to process than uncompressed text characters: They are bit-codings of move-to-front values. Instead of just one comparison and an optional increment over registers, we have to (1) extract the bits, (2) obtain their numerical value from the Huffman tree, (3) search the move-to-front linked list (usually a few positions, say 3 or 4), (4) move the element found to front, and (5) do the comparison and optional increment as well. Decoding move-to-front is the slowest part of this picture, which can perfectly exceed the 12- or 20-fold gain. Indeed, in our experiments, processing each compressed megabyte took 387 μsec on ZIFF and 338 on DNA. Comparing and adding took 4.12 μsec on ZIFF and 2.06 on DNA. This shows that decompression poses a 100- to 150-fold slowdown, much larger than the 12- to 20-fold speedup.

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

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

Our implementation of the FM-index can be obtained freely from `http://www.dcc.uchile.cl/gnavarro/software`.

12.3 Sadakane's CSArray

We obtained Sadakane's [2000] implementation of the Compressed Suffix Array index.[17] This implements a suffix array search over a compressed representation of it. The main data structures are Ψ, which permits moving from the suffix array position that points to i to that pointing to $i+1$, and a sampling of the suffix array. The array Ψ can be stored with a technique similar to that used by the FM-index to store the text, as values $\Psi(i) - \Psi(i-1)$ tend to be small. It also needs some directory structures to compute cumulative frequencies, just as the FM-index. The sampling of the suffix array plays a similar role as well. As can be seen, the ideas are not so radically different after some analysis. This index permits moving "forward" in the text, while the FM-index moves "backward."

[17]Currently available in the *Pizza&Chili* site.

Table II. Index Construction Requirements.
Times are in seconds per MB and space in number of times the text size

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

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

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

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

12.4 Comparison

Recall that we compare the three indexes such that they take the same amount of main memory to operate. Table II shows the time and memory requirements to build the different indexes (although the final index space is the same, they need different space to build). As it can be seen, our index (LZ-index) builds much faster than the others (whose construction time involve at least the construction of a suffix array). It also needs less memory to build. We note that the original implementation of the FM-index (by its authors) builds faster than ours (2.257 seconds for ZIFF and 1.704 for DNA, using 9.000 times the text size to build), but still significantly slower than the LZ-index.

Let us now consider search times. Figure 24 shows the overall query times under the different reporting levels. Note that we use a logarithmic scale on y.

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

The FM-index, however, becomes *much* slower to report the positions of the occurrences found, achieving a rate of 10–20 occurrences per msec (recall that

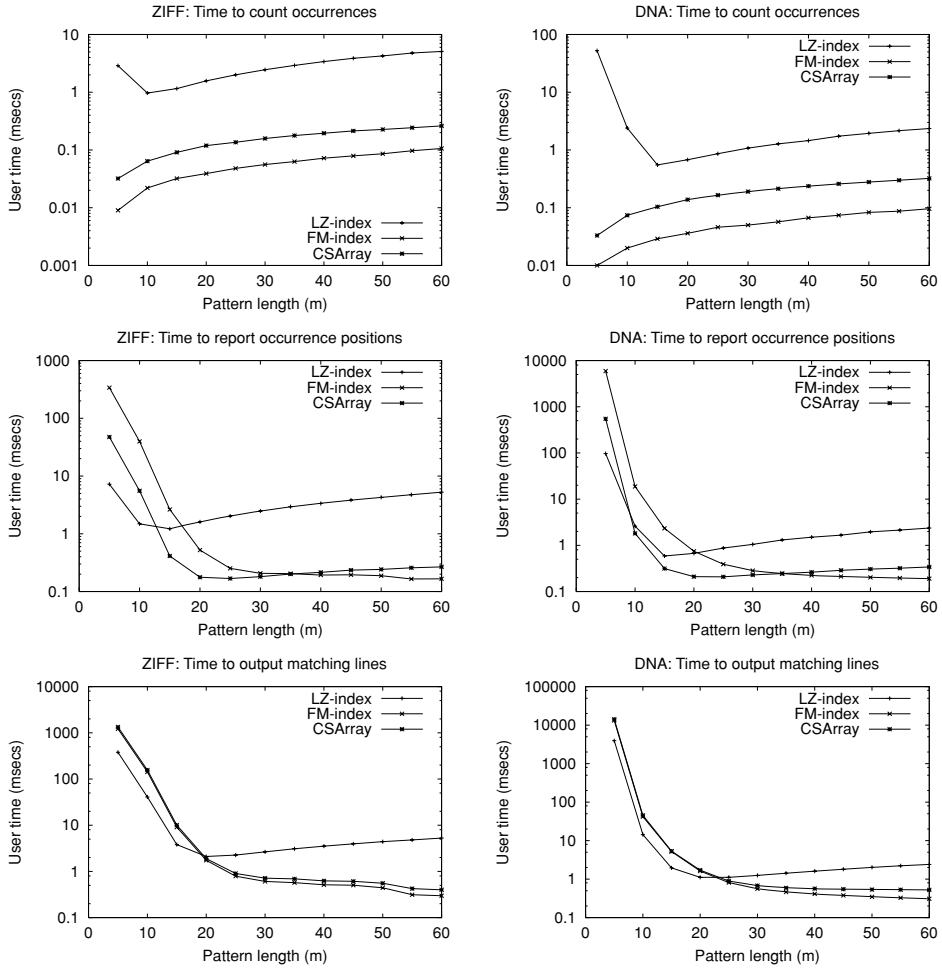

Fig. 24. Overall query times when counting occurrences (top), reporting positions (middle), and to output matching lines (bottom). We compare our LZ-index against the most relevant alternatives.

our rate is closer to 900 to 1,400 per msec, depending on the hit ratio on candidates of type 2). The CSArray is faster than the FM-index at this step, reporting 100 to 160 occurrences per msec, by taking advantage of the denser sampling. This is, in turn, a consequence of the different compression methods used. In any case, it is clear that finding the actual positions of the occurrences is costly under their schemes, 70 to 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. Remember 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 to 6 lines per msec. This time the sampling of the suffix array plays a less important role (and hence

the FM-index and CSArray implementations have little differences). The reason is that the sampled suffix array is used just to find the occurrence position, and from then on one letter is output per step (forward or backward move in the original text).

Summarizing, 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, instead, rely on a sampled suffix array and must perform expensive traversals until they determine the actual suffix array values.

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

Let us take the best from both competing indexes. Assume 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. It turns out that, to report the occurrences, the LZ-index would become faster after we report 1,400 occurrences on ZIFF or 300 on DNA. If we would like to see the lines containing the occurrences, these numbers drop to 65 on ZIFF and 13 on DNA. This shows that our index becomes superior as soon as we have to show a few occurrences.

To conclude, we give some data on our tests over the executables of the FM-index provided by the authors. These permit a coarse control over the index space by specifying the frequency of a character whose positions will be sampled. Although we tried the highest possible frequencies, we could not obtain indexes larger than 75.02% of the ZIFF file and 109.81% of the DNA file. The former is half the space we permit, while the latter is rather close to the correct value. The main memory required to build the index is 9 times the text size, and the construction speed is 1.7 to 2.3 sec/MB. The time to count occurrences is negligible, as expected. Occurrence positions were reported at a rate that varied a lot but was always between 0.5 and 10 occurrences per msec. When we asked the index to show a text context of length equivalent to an average line (43 characters on ZIFF and 61 on DNA), it showed them at a rate of 10 to 20 per *second*. Even if we assume that the index on ZIFF could double its performance by using twice the space, the figures still show that our implementation of the FM-index is competitive against that of the original authors, when not superior by far.[18] The results did not vary by trying different memory policies offered by the index (on disk, mmaped, or in main memory).

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

13. CONCLUSIONS

We have presented the process of obtaining a real implementation for the LZ-index, a succinct data structure for text searching which existed as a theoretical proposal. We have considered the tradeoffs between theoretically appealing ideas and practically efficient implementations. We have empirically studied the behavior of the many aspects of our index and shown that the average search cost of our implementation is of the form $O(\sigma m \log u + \sqrt{uR})$, where u is the text size, m the pattern length, σ the alphabet size, and R the number of occurrences. A prototype of our index is available at `http://www.dcc.uchile.cl/gnavarro/software`.

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

In our way, we have learned several lessons of theory versus practice, for example: (1) the traps of asymptotic analyses that do not become realistic for the text sizes we expect to handle in this decade, with asymptotically vanishing terms that are huge in practice (e.g., balanced parentheses); (2) the traps of extremely complicated proposals that are much easier to write in paper than to implement (e.g., the *LZTrie* implementation); (3) the importance of cache- and CPU register-awareness when comparing the cost of alternatives of similar complexities (e.g., the lowest level of bitmaps with $rank$); (4) the unpracticality of schemes that give worst case guarantees but are much slower on average or in practice than others that do not (e.g., the *Range* data structure); (5) the risk of theoretically perfect techniques that in practice work only for toy examples (e.g., perfect hashing); (6) the importance of being realistic in the analysis of small complexities (e.g., our $O(\log \log n)$ solutions, that in practice are as good as constant time ones); and (7) the large influence of factors that are completely outside of the algorithmic design, such as the cost of freeing a scattered data structure (e.g., freeing the pointer-based tries).

The previous paragraph does not mean that theoretical developments should be disregarded. Much to the contrary, they are the essential core of the progress in algorithmics. For example, the success of our index owes to its theoretical development and to other brilliant theoretical solutions to handle bitmaps and general trees [Munro and Raman 1997; Munro et al. 2001]. What we mean is that, in many cases, one cannot hope to implement theoretical solutions "as-is," but rather be prepared to take their high-level concepts (which are usually the most valuable) and redesign the algortihmic details to match practical cost criteria. Of course, there are theoretical developments that are directly applicable, as well as others that cannot be applied at all.

We have also seen the importance of coupling the empirical and theoretical analyses in order to fully understand the behavior of the structures we design and to make the right design decisions. Furthermore, the design and the analysis must be parallel processes influencing each other.

Finally, we have observed that much of the efficiency in a real implementation of the competing indexes may have to do with aspects that are usually disregarded at the algorithmic level, such as the exact way the compression is performed.

As a final followup note, we mention further developments on the LZ-index. New variants have been designed that are able of running in up to half the space while *improving* the search complexity [Arroyuelo et al. 2006; Arroyuelo and Navarro 2007b], of building the LZ-index within basically the same space needed by the final index [Arroyuelo and Navarro 2005], and working efficiently on secondary memory [Arroyuelo and Navarro 2007a]. A range of space/time tradeoff variants have been implemented and are available at the *Pizza&Chili* site, where a technical report compares one of those versions against the most recent alternatives. The conclusions we draw in this article about the suitability of the LZ-index against other compressed text indexes stay roughly the same. Other variants have also appeared, such as the ILZI [Russo and Oliveira 2006], where the ability of carrying out complex searches has been explored [Russo et al. 2007]. Nowadays, as the theoretical versus practical gap in compressed text indexing research has been considerably reduced, the active areas of research are shifting to construction in little space, handling secondary memory, handling changes in the text collection, and supporting complex searches. We expect the research in the next years to go in those directions, hopefully combining theory and practice.

REFERENCES

Abouelhoda, M., Ohlebusch, E., and Kurtz, S. 2002. Optimal exact string matching based on suffix arrays. In *Proceedings of the 9th International Symposium String Processing and Information Retrieval (SPIRE)*. LNCS 2476. 31–43.

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

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

Arroyuelo, D. and Navarro, G. 2005. Space-efficient construction of LZ-index. In *Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC)*. LNCS 3827. 1143–1152.

Arroyuelo, D. and Navarro, G. 2007a. A lempel-ziv text index on secondary storage. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4580. 83–94.

Arroyuelo, D. and Navarro, G. 2007b. Smaller and faster lempel-ziv indices. In *Proceedings of the 18th International Workshop on Combinatorial Algorithms (IWOCA)*. College Publications, UK, 11–20.

Arroyuelo, D., Navarro, G., and Sadakane, K. 2006. Reducing the space requirement of LZ-index. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4009. 319–330.

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

BENOIT, D., DEMAINE, E., MUNRO, I., RAMAN, R., RAMAN, V., AND RAO, S. 2005. Representing trees of higher degree. *Algorithmica 43*, 4, 275–292.

CHAZELLE, B. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing 17*, 3, 427–462.

FERRAGINA, P. AND MANZINI, G. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium Foundations of Computer Science (FOCS)*. 390–398.

FERRAGINA, P. AND MANZINI, G. 2001. An experimental study of an opportunistic index. In *Proceedings of the 12th ACM Symposium on Discrete Algorithms (SODA)*. 269–278.

FERRAGINA, P. AND MANZINI, G. 2002. On compressing and indexing data. Tech. Rep. TR-02-01, Dipartamento di Informatica, Univ. of Pisa.

GEARY, R., RAHMAN, N., RAMAN, R., AND RAMAN, V. 2004. A simple optimal representation for balanced parentheses. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS v. 3109. 159–172.

GONZÁLEZ, R., GRABOWSKI, S., MÄKINEN, V., AND NAVARRO, G. 2005. Practical implementation of rank and select queries. In *Poster Proceedings Volume of the 4th Workshop on Efficient and Experimental Algorithms (WEA)*. CTI Press and Ellinika Grammata, Greece, 27–38.

GROSSI, R. AND VITTER, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium Theory of Computing (STOC)*. 397–406.

HARMAN, D. 1995. Overview of the Third Text REtrieval Conference. In *Proceedings of the 3rd Text REtrieval Conf. (TREC-3)*. 1–19. NIST Special Publication 500-507.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium Foundations of Computer Science (FOCS)*. 549–554.

KÄRKKÄINEN, J. 1995. Suffix cactus: a cross between suffix tree and suffix array. In *Proceedings of the 6th Annual Symposium Combinatorial Pattern Matching (CPM)*. LNCS 937. 191–204.

KÄRKKÄINEN, J. 1999. Repetition-based text indexes. Ph.D. thesis, Dept. of Computer Science, University of Helsinki, Finland. Also available as Report A-1999-4, Series A.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996a. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing (WSP)*. Carleton University Press, 141–155.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996b. Sparse suffix trees. In *Proceedings of the 2nd Ann. International Conf. on Computing and Combinatorics (COCOON)*. LNCS 1090.

KOSARAJU, R. AND MANZINI, G. 1999. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing 29*, 3, 893–911.

KURTZ, S. 1998. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Universität Bielefeld.

MÄKINEN, V. 2003. Compact suffix array—a space-efficient full-text index. *Fundamenta Informaticae 56*, 1–2, 191–210.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 935–948.

MUNRO, I. 1996. Tables. In *Proceedings of the 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 1180. 37–42.

MUNRO, I. AND RAMAN, V. 1997. Succint representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th IEEE Symposium Foundations of Computer Science (FOCS)*. 118–126.

MUNRO, I., RAMAN, V., AND RAO, S. 2001. Space efficient suffix trees. *Journal of Algorithms*, 205–222.

NAVARRO, G. 2002. Indexing text using the Ziv-Lempel trie. In *Proceedings of the 9th International Symposium String Processing and Information Retrieval (SPIRE)*. LNCS 2476. 325–336.

NAVARRO, G. 2004. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA) 2*, 1, 87–114.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39*, 1, article 2.

NAVARRO, G., MOURA, E., NEUBERT, M., ZIVIANI, N., AND BAEZA-YATES, R. 2000. Adding compression to block addressing inverted indexes. *Information Retrieval 3*, 1, 49–77.

Russo, L., Navarro, G., and Oliveira, A. 2007. Approximate string matching with Lempel-Ziv compressed indexes. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 4726. 265–275.

Russo, L. and Oliveira, A. 2006. A compressed self-index using a ziv-lempel dictionary. In *Proceedings of the 13th Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 4209. 163–180.

Sadakane, K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th International Symposium Algorithms and Computation (ISAAC)*. LNCS 1969. 410–421.

Sadakane, K. 2002. Succint representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the 13th ACM Symposium on Discrete Algorithms (SODA)*. 225–232.

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

Witten, I., Moffat, A., and Bell, T. 1999. *Managing Gigabytes*, second ed. Morgan Kaufmann Publishers, New York.

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

Received January 2003; accepted June 2003