

# Opportunistic data structures for range queries\*

Chung Keung Poon · Wai Keung Yiu

Received: 25 September 2005 / Accepted: 25 December 2005  
© Springer Science + Business Media, LLC 2006

**Abstract** In this paper, we study the problem of supporting range sum queries on a compressed sequence of values. For a sequence of  $n$   $k$ -bit integers,  $k \leq O(\log n)$ , our data structures require asymptotically the same amount of storage as the compressed sequence if compressed using the Lempel-Ziv algorithm. The basic structure supports range sum queries in  $O(\log n)$  time. With an increase by a constant factor in the storage complexity, the query time can be improved to  $O(\log \log n + k)$ .

## 1. Introduction

With the proliferation of electronic data nowadays, there is a growing demand to store data in compressed form. In fact, data compression has long been recognized as an important area in computer science and engineering. Numerous compression algorithms (Huffman, 1952; Ziv and Lempel, 1978; Golomb, 1966; Burrows and Wheeler, 1994) have been devised and used in day-to-day computer operations to reduce the storage requirement for data as well as the necessary bandwidth for data transmission.

Very often, we need to retrieve or operate on only part of the data. It would be desirable if the data is compressed in such a way that one could inspect or extract information about part of the original data directly from the compressed data. For example, textual information can be stored and indexed by a compressed suffix array (Grossi and Vitter, 2000; Sadakane, 2003, 2002) or by Burrows-Wheeler Transform combined with Lempel-Ziv algorithm (Ferragina and Manzini, 2002) so that one can search for a pattern in the text efficiently. However, for many fundamental data structural problems including range sum queries, we are not aware of any focused research reported in

---

\*The work described in this paper is fully supported by a grant from the Research Grant Council of the Hong Kong Special Administrative Region, China (CityU 1071/02E). A preliminary version has appeared in 11th International Conference in Computing and Combinatorics (COCOON'05).

the literature. Thus we are motivated to study the range sum query problems in this paper.

Formally, our problem can be defined as follows. Given an array  $A[0..n - 1]$  of  $k$ -bit integers,  $k \leq O(\log n)$ , compress it so that range sum queries, i.e., the sum of values in  $A[i..j]$  given the boundaries  $i$  and  $j$ , can be answered efficiently. Note that when  $i = j$ , the range query becomes a point query. That means, the data structure should be able to report every entry of  $A$  without decompressing the whole data. Clearly, the major performance measures of interest are the storage and query complexities. We will assume a unit-cost word RAM with word size  $\Theta(\log n)$ . On such a model, standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time. Throughout this paper, storage complexities are expressed in terms of bits.

A simple approach to solve the problem is to compute the prefix (or partial) sum array  $P[0..n - 1]$  such that  $P[i]$  stores the sum of  $A[0..i]$ . Any range sum,  $sum(A[i..j])$ , can be computed in  $O(1)$  time by taking the difference between  $P[j]$  and  $P[i - 1]$ . Storing  $P[0..n - 1]$  requires  $O(n \log n)$  bits. This storage requirement can be reduced by way of a succinct (or space-efficient) data structure. The aim of such a structure is to achieve optimal space usage to within lower order additive terms while having asymptotically optimal operation times. In particular, using structures by Raman et al. (2001) and Hon et al. (2003) with suitable parameters, the partial sum problem can be solved in  $O(1)$  time and  $kn + o(kn)$  bits.

However, a succinct data structure is still not taking the full opportunity offered by the particular data stored in  $A$ . To illustrate this point, suppose there are only  $m \ll n$  non-zero entries in  $A$ . Then, we can store the indices to those non-zero entries in  $A$  together with the associated prefix sums in a predecessor structure. Computing a range sum queries then amounts to two predecessor lookups and an integer subtraction. Using the linear space predecessor structure of Willard (1983), it takes  $O(\log \log n)$  time and  $O(m \log n)$  storages.

In general, the array  $A$  may or may not contain many zeroes. Thus committing ourselves to either a predecessor structure or a succinct data structure without prior knowledge of  $A$  may not work well always. Even if we do a scan on  $A$  before making the choice, it could happen that  $A$  contains few zeroes but is nevertheless rather compressible. To take full advantage of the compressibility of the data, we propose to design data structures whose storage is asymptotically the same as the size of the compressed data while having the same operation times as in a traditional or succinct data structure.

For the range sum query problem here, we design data structures whose space requirement is of the same order as the compressed array when compressed using the Lempel-Ziv algorithm (Ziv and Lempel, 1978) (commonly known as *ZL78*). Our basic structure answers a range sum query in  $O(\log n)$  time. With an increase by a constant factor in storage complexity, we obtain a variant in which the query time is  $O(\log \log n + k)$ . Also our method does not modify the compressed data and can be viewed as an additional index structure on it.

Our method is based on the Lempel-Ziv algorithm (obviously) and employed many standard techniques in succinct data structures. The Lempel-Ziv algorithm and its variants are an important class of compression algorithms. Its behaviour is well-understood and is optimal in certain information-theoretic sense. It is popular and easy to implement. For example, it is implemented in the *compress* program in UNIX and in the *arc* program for PC's. In the next section, we will describe the main idea of the Lempel-Ziv algorithm. Then we explain our method for bit strings in sections 3. Section 4 describes the extension to general arrays and some variations. Section 5 contains the conclusion.

## 2. The Lempel-Ziv’s algorithm

The Lempel-Ziv algorithm (Ziv and Lempel, 1978) is a lossless compression algorithm that will automatically adapt to the data distribution. It can be applied to strings over a finite alphabet. For our usage here, we just need to understand the compression and we will omit the decompression. We will first describe the idea on bit strings.

### 2.1. Parsing the string

The idea is to partition the bit string into *phrases* not appeared before. After marking off the end of the last phrase, we start from the next bit in the input sequence until we come to the shortest string  $s$  that has not been marked off before. Denote by  $s^-$  the longest prefix of  $s$ , i.e., all but the last bit of  $s$ . By minimality of  $|s|$ ,  $s^-$  has appeared as a phrase before. We mark off  $s$  as a new phrase and encode it by the pair  $(p, b)$  where  $p$  is the index of the phrase  $s^-$  (stored in a dictionary of discovered phrases, to be described in section 2.2) and  $b$  is the last bit of  $s$ . Note that by construction,  $s^-$  appears only once before. Moreover, each phrase (of variable length) is now encoded as a fixed length code.

**A running example:** Consider the following input string. (Spaces are inserted in the string for clarity.)

1 0 01 10 011 11 010 101 00 011

It will be parsed into the following sequence of phrases:

*abc def ghi e*

where the substring represented by each phrase is shown in Figure 1.

Note that only the last phrase can be a repetition of some previous phrase. The encoded string is

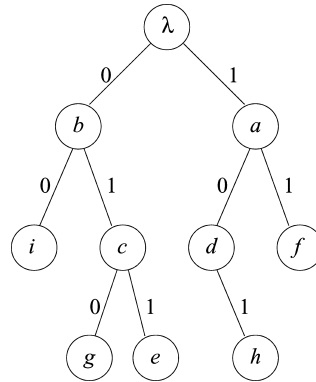
$\lambda 1 \lambda 0 b 1 a 0 c 1 a 1 c 0 d 1 b 0 e$

where  $\lambda$  represents the empty string. If there are altogether  $c(n)$  distinct phrases, each index  $p$  can be encoded in  $\log c(n)$  bits. In total, the encoded string has length  $c(n)(\log c(n) + 1)$  bits (or  $(c(n) + 1)(\log c(n) + 1) - 1$  bits if the last phrase has appeared before). It takes  $O(c(n) \log c(n))$  bits asymptotically in both cases.

Plugged into the example in Figure 1, the original string has 22 bits. There are 10 phrases in the encoded string, the last one being a repetition of another phrase before. Therefore, the encoded string has  $9 \times (4 + 1) + 4 = 49$  bits which is longer than the original string. For longer strings with many repeating patterns, the phrases will get longer and encoding a long phrase with  $\log c(n)$  bits will become a big saving. More rigorously, how good we can compress with Lempel-Ziv is controlled by the parameter  $c(n)$ . It can be shown that  $\sqrt{n} \leq c(n) \leq O(n / \log n)$ . Moreover, if we assume the values of  $A$  are drawn from a stationary ergodic source with entropy rate  $H$  (i.e., it takes on average  $H$  bits to describe one value

phrase	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>
substring represented	1	0	01	10	011	11	010	101	00

**Fig. 1** The string represented by each phrase

Fig. 2 Trie  $T$ 

of  $A$  in the long run), then  $\frac{c(n) \log c(n)}{n} \rightarrow H$  as  $n \rightarrow \infty$ . See, for example, Lemma 12.10.1 and Theorem 12.10.1 of Cover and Thomas (1991) for more details.

## 2.2. A dictionary of phrases

During the parsing of the input sequence, a dictionary of phrases is gradually built up to facilitate the discovery of new phrases. Newly discovered phrases will be added to the dictionary. An appropriate data structure for the dictionary is the binary trie structure.

A binary trie is a tree in which each internal node has at most two children. The edge to the left (resp. right) child will be labelled with 0 (resp. 1). Each node in the trie corresponds to a string, namely, the string obtained by concatenating all the edge labels in the order from the root to that node. As a special case, the root corresponds to the empty string.

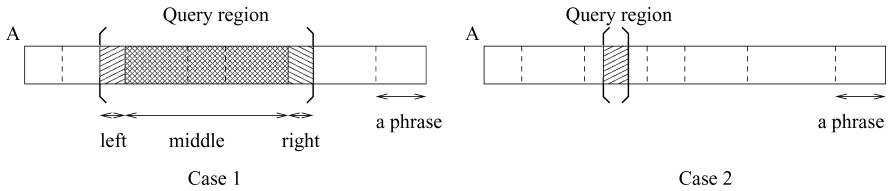
The Lempel-Ziv algorithm will construct a trie  $T$  so that each node corresponds to a phrase discovered in the input sequence scanned so far. When we try to discover a new phrase in the remainder of the input sequence, we search  $T$  from the root and follow the edges as we read off the bits from the input sequence until we reach a leaf. Then the string formed by appending the next bit in the input sequence to the string represented by that leaf is a new phrase not yet appeared before. We add this to the trie by creating a left or right child to this leaf depending on the next bit is 0 or 1. Thus, parsing requires  $O(n)$  time. For our running example, the corresponding trie  $T$  is shown in Figure 2.

In the ordinary usage of Lempel-Ziv,  $T$  is discarded after parsing is completed. Here, we will keep  $T$  in order to facilitate the construction of our data structure. After the construction,  $T$  is discarded.

## 3. Range sum queries on compressed bit strings

Now we are ready to describe our data structure for bit strings. On the highest level, we partition the input bit string into phrases according to the Lempel-Ziv parsing mentioned in section 2.1. Then we will build two structures:

- the *inter-phrase structure* that supports queries on the number of 1's in a contiguous sequence of phrases, and
- the *intra-phrase structure* that supports queries on the number of 1's in a continuous range within a phrase.



**Fig. 3** Breaking down a query region

In general, a query range covers a (possibly zero) number of phrases completely and at most two phrases partially. Thus a query range can be broken into at most three parts. See Figure 3. The sequence of completely covered phrases is handled using the inter-phrase structure while the partially covered phrases are handled using the intra-phrase structure.

### 3.1. The inter-phrase structure

The inter-phrase structure contains the arrays  $P$  and  $S$ . In  $P$ , we store in ascending order the starting position of each phrase in the input sequence and in  $S$ , we store the number of 1’s in front of each phrase. See Figure 4 for an illustration. In total,  $P$  and  $S$  require  $O(c(n) \log n)$  space and can be constructed in  $O(n)$  time during the parsing of the input sequence.

Given a query region  $[\ell, r]$ , we find the smallest  $i$  such that  $P[i] \geq \ell$  and the largest  $j$  such that  $P[j] - 1 \leq r$ . This takes  $O(\log c(n)) = O(\log n)$  time by binary search on  $P$ . Then the  $i$ -th to  $(j - 1)$ -st phrases are completely within the query range. The number of 1’s in this sequence of phrases is computed as  $S[j] - S[i]$ .

After computing  $i$  and  $j$  in the above query, we need to query the intra-phrase structure to determine the number of 1’s in a range in the  $(i - 1)$ -st and the  $j$ -th phrase. To allow for locating any desired phrase in the intra-phrase structure, the inter-phrase structure will also store an array of pointers (bottom row in Figure 4). It will be clear in the next section that each pointer requires  $O(\log c(n))$  bits. Hence the array of pointers requires  $O(c(n) \log c(n)) = O(c(n) \log n)$  bits. This array is constructed at the same time when the inter-phrase structure is being built and it takes  $O(c(n))$  time.

### 3.2. The intra-phrase structure

To compute the number of 1’s within a range in a phrase quickly, we make use of the trie  $T$ . In what follows, we will describe a succinct representation of  $T$  in  $4c(n) + o(c(n))$  bits, using many standard techniques in succinct data structure design. This will be useful in minimizing the storage in practical implementations. We begin with a few easy definitions.

*Definition 1.* The *depth* of a node in the trie is defined as the number of edges from the root to that node.

*Definition 2.* The *level- $d$  ancestor* of a node  $v$  is the (unique) node that has depth  $d$  on the path from the root to  $v$ .

**Fig. 4** Inter-phrase structure

$P$	0	1	2	4	6	9	11	14	17	19
$S$	0	1	1	2	3	5	7	8	10	10
pointer to phrase	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$	$e$

*Definition 3.* The *Euler tour* of a rooted tree  $T$  is the sequence of edges traversed during a depth-first traversal of  $T$  starting from the root, and ending at the root after visiting all the nodes in  $T$ .

For example, the Euler tour of our trie  $T$  is shown on the row labelled with  $E$  in Figure 5. Here, we represent an edge by one of the 4 symbols,  $(_0, (1, )_0$  and  $)_1$ , depending on whether the edge is downward or upward and with label 0 or 1. Since each edge is traversed twice, this sequence contains  $2c(n)$  brackets in total. Furthermore, each bracket is associated with a node, namely, the node reached by following that edge. See the row labelled with “phrase” in Figure 5.

Notice that a node may appear more than once in the Euler tour.

*Definition 4.* An *index* of a node  $v$  in a tree  $T$  is the position of an occurrence of  $v$  in the Euler tour of  $T$ , and the *principle index* is the position of the leftmost occurrence.

The intra-phrase structure will support the following primitive operations:

*Count*( $v$ ) – Given an index of a node  $v$ , find the number of 1’s on the path from the root to  $v$ .

*Depth*( $v$ ) – Given an index of a node  $v$ , find the depth of  $v$ , i.e., the number of edges from the root to  $v$ .

*Ancestor*( $v,d$ ) – Given the principle index of a node  $v$  and an integer  $d$ , find an index of the level- $d$  ancestor of  $v$ .

Given these three operations, we can compute the number of 1’s on the path from the level- $i$  ancestor to the level- $j$  ancestor of a node  $v$ , given  $i, j$  and the principle index of node  $v$ . That is, we compute  $Count(Ancestor(v,j)) - Count(Ancestor(v,i-1))$ . This corresponds to the number of 1’s in the continuous range from position  $i$  to  $j$  of a phrase  $v$ . To be able to invoke *Ancestor*( $v,d$ ), the inter-phrase structure stores the principle indices of the phrases in the array of pointers (bottom row of Figure 4). Each index requires  $O(\log c(n))$  bits.

To support the three primitive operations, we store two arrays,  $C$  and  $D$ , of bits. See Figure 5. (Note that we do not store  $E$  and the row labelled with “phrase”.) The array  $D$  will give information on the depth of the nodes in the trie while  $C$  and  $D$  together will give information on the number of 1’s along the path from the root to each node.

More precisely, for  $0 \leq i < 2c(n)$ , we define  $D[i]$  as +1 (−1) if  $E[i]$  is an open (close) bracket. Hence, the sum of values in  $D[0..i]$  is the difference between the number of open and close brackets in  $E[0..i]$ . This, in turn, represents the depth of the node reached by following the traversal specified in  $E[0..i]$ . Similarly, for  $0 \leq i < 2c(n)$ , we define  $C[i]$  as

**Fig. 5** Intra-phrase structure

phrase	$b$	$i$	$b$	$c$	$g$	$c$	$e$	$c$	$b$
$E$	$(_0$	$(_0$	$)_0$	$(1$	$(_0$	$)_0$	$(1$	$)_1$	$)_1$
$D$	+1	+1	−1	+1	+1	−1	+1	−1	−1
$C$	−1	−1	+1	+1	−1	+1	+1	−1	−1

phrase	$\lambda$	$a$	$d$	$b$	$d$	$a$	$f$	$a$	$\lambda$
$E$	$)_0$	$(1$	$(_0$	$(1$	$)_1$	$)_0$	$(1$	$)_1$	$)_1$
$D$	−1	+1	+1	+1	−1	−1	+1	−1	−1
$C$	+1	+1	−1	+1	−1	+1	+1	−1	−1

$$C[i] = \begin{cases} +1 & \text{if } E[i] = )_0 \text{ or } ( _1 \\ -1 & \text{if } E[i] = )_1 \text{ or } ( _0 \end{cases} .$$

It is easy to check that

$$C[i] + D[i] = \begin{cases} +2 & \text{if } E[i] = ( _1 \\ -2 & \text{if } E[i] = )_1 \\ 0 & \text{if } E[i] = )_0 \text{ or } ( _0 \end{cases} .$$

Hence the sum of values in  $C[0..i]$  and  $D[0..i]$  gives twice the number of 1’s from the root to the node corresponding to  $E[i]$ .

### 3.3. Supporting $Count(v)$ and $Depth(v)$

We need to compute the sum of values in  $C[0..i]$  and  $D[0..i]$  where  $i$  is an index of  $v$  in  $E$ . We will explain the computation for  $C[0..i]$ . (Computation for  $D[0..i]$  is identical.) The technique is typical in succinct data structures.

We will construct an array  $C^0$  with  $\frac{2c(n)}{\log^2 c(n)} = o(c(n))$  entries such that for  $i = 0$  to  $\frac{2c(n)}{\log^2 c(n)} - 1$ , the entry  $C^0[i]$  will store the sum in  $C[0..(i + 1)\log^2 c(n) - 1]$ . Since  $C$  has length  $2c(n)$ , the maximum sum can be stored in  $O(\log c(n))$  bits. Hence  $C^0$  will occupy  $O\left(\frac{c(n)}{\log^2 c(n)} \times \log c(n)\right) = o(c(n))$  bits. We can, in  $O(1)$  time, look up the number of 1’s in  $C[0..i]$  when  $i$  is a multiple of  $\log^2 c(n)$ .

Next, we will construct another array  $C^1$  with  $\frac{4c(n)}{\log c(n)}$  entries. For  $i = 0$  to  $\frac{2c(n)}{\log^2 c(n)} - 1$  and for  $j = 0$  to  $2\log c(n) - 1$ , the entry  $C^1[i \cdot 2\log c(n) + j]$  will contain the sum in  $C\left[i \log^2 c(n) \dots i \log^2 c(n) + (j + 1)\frac{\log c(n)}{2} - 1\right]$ . Since each short-ranged sum is at most  $\log^2 c(n)$ , each entry of  $C^1$  requires only  $\log(\log^2 c(n))$  bits. Hence  $C^1$  occupies  $O\left(\frac{4c(n)}{\log c(n)} \times \log \log c(n)\right) = o(c(n))$  bits. With  $C^1$  and  $C^0$ , we can compute the sum in  $C[0..i]$  when  $i$  is a multiple of  $\frac{\log c(n)}{2}$ .

Finally, for the number of 1’s within a  $(\pm 1)$ -pattern of length  $\frac{\log c(n)}{2}$ , we make use of a 2-dimensional lookup table  $L$ . Notice that there are  $2^{(\log c(n))/2} = \sqrt{c(n)}$  different  $(\pm)$ -patterns of length  $\frac{\log c(n)}{2}$ . For each such pattern  $\alpha$  and for each position  $0 \leq j < \frac{\log c(n)}{2}$ , the sum of  $\pm 1$ ’s from position 0 to  $j$  can be precomputed and stored in  $L[\alpha, j]$ . Thus,  $L$  has size  $\sqrt{c(n)} \times \frac{\log c(n)}{2} \times \log \log c(n) = o(c(n))$  bits. Moreover, each chunk of  $\log c(n)/2$  bits in  $C$  can share the same table  $L$ .

We will similarly construct  $D^0$  and  $D^1$  for  $D$  but we can share the same lookup table  $L$ . In total, the storage required by the arrays  $C, C^0, C^1, D, D^0, D^1$  and  $L$  is  $4c(n) + o(c(n))$  bits. Moreover, the operations require only constant time to complete.

### 3.4. Supporting $Ancestor(v, d)$

Observe that the lowest common ancestor (LCA) of  $v$  with any node is an ancestor of  $v$ . By the property of an Euler tour, the LCA of the nodes with indices  $x$  and  $y$  ( $x \leq y$ ) is the node of minimum depth among those with an index  $z$  such that  $x \leq z \leq y$ . Thus, if  $j$  is the principle index of  $v$ , then for any  $i \leq j$ , the LCA of node  $v$  and the node corresponding to  $i$  is the node with minimum depth in  $E[i..j]$ . Since the minimum value in  $E[i..j]$  is no less than that in  $E[i'..j]$  for any  $i' \leq i$ , the depth of the LCA between the nodes with indices  $i$  and  $j$  is monotonic increasing as  $i$  increases from 0 to  $j$ .

Therefore the level- $d$  ancestor of  $j$  can be found by a binary search in  $O(\log c(n)) = O(\log n)$  time, provided we can compute the depth of the LCA between any pair of indices,  $(i, j)$ , in constant time. This amounts to computing the minimum among  $D[0..i]$ ,  $D[0..i + 1]$ ,  $\dots$ ,  $D[0..j]$ . In the next subsection, we will describe a data structure of space  $o(c(n))$  that answers such implicit range minimum queries in  $O(1)$  time. Note that there exist algorithms that support level ancestor query in  $O(1)$  time using space  $O(n \log n)$  (Bender and Farach-Colton, 2000) or even  $O(n)$  (Geary et al., 2004). However, we do not use them here because the overall query time is not improved and they do not blend well with the other components of our structures, resulting in a larger constant factor in the storage.

### 3.5. A range minimum structure

Let  $\tilde{D}_1$  be the array storing the index to the minimum of each sub-interval of length  $\log^3 c(n)$  in  $D$ . Thus  $\tilde{D}_1$  has  $2c(n)/\log^3 c(n)$  entries, each storing an  $O(\log c(n))$ -bit index. On this array, we build an APM structure of Poon (2003) which can answer range min queries in constant time and uses  $\frac{2c(n)}{\log^3 c(n)} \log\left(\frac{2c(n)}{\log^3 c(n)}\right) \log c(n) = o(c(n))$  bits. We denote this structure by  $APM(\tilde{D}_1)$ .

For each interval of length  $\log^3 c(n)$  in  $D$ , we further break it down into sub-intervals of length  $\log c(n)/2$  and store the index to the minimum of each sub-interval in another array  $\tilde{D}_2$  of length  $2c(n)/\log^3 c(n) \times 2\log^2 c(n) = 4c(n)/\log c(n)$ . This time, each index will need only  $O(\log \log c(n))$  bits since they are indices relative to a sub-interval. Again, we build an APM structure for every  $2\log^2 c(n)$  entries of  $\tilde{D}_2$ . We denote this collection of structures by  $APM(\tilde{D}_2)$ . This requires  $\frac{2c(n)}{\log^3 c(n)} (2\log^2 c(n) \log(2\log^2 c(n)) \log \log c) = o(c(n))$  bits.

Finally, we build a lookup table  $M$  for range minimum within a sub-interval of length  $\log c(n)/2$ . The table has size  $(\sqrt{c(n)}) \times (\log c(n)/2)^2 \times \log \log c(n)$  bits. The table is shared by all the sub-intervals of length  $(\log c(n))/2$ . In total, the two structures,  $APM(\tilde{D}_1)$  and  $APM(\tilde{D}_2)$ , together with  $M$ , require at most  $o(c(n))$  bits.

## 4. Extension and variations

### 4.1. Generalizing to array of integers

To extend the previous ideas on a general array of  $k$ -bit integers, we apply Lempel-Ziv over an alphabet of size  $2^k$ . Thus, the dictionary of phrases  $T$  will be a  $2^k$ -ary trie instead of a binary trie.

For the inter-phrase structure, we have  $P$ ,  $S$  and the array of principle indices using  $O(c(n) \log n)$  bits of storage. For the intra-phrase structure, we store  $T$  succinctly using array  $D$  as defined before but with arrays  $C_0, C_1, \dots, C_{k-1}$  defined as follows. For each  $i \in \{0, \dots, k-1\}$ ,  $C_i$  is designed such that the sum of values in  $C_i[0..j]$  and  $D[0..j]$  is equal to twice the number of 1's in the  $i$ -th bit position of the phrase from the root to the node corresponding to  $E[i]$ . Furthermore, we will store the succinct data structure for level-ancestor  $APM(\tilde{D}_1)$  and  $APM(\tilde{D}_2)$ . In total, these require  $(2(k+1)c(n) + o(c(n))) = O(c(n) \log n)$  bits of storage as  $k = O(\log n)$ .

To handle a query, we first query the inter-phrase structure as in section 3.1. For the intra-phrase query, we compute the number of 1's within the required range in each of the  $k$  bit

positions. Then we shift these  $k$  sums properly and sum them. More precisely, the sum of values between position  $d_1$  and  $d_2$  (inclusive) within a phrase  $v$  is calculated as:

$$\sum_{i=0}^{k-1} 2^i (\text{Count}_i(\text{Ancestor}(v, d_2)) - \text{Count}_i(\text{Ancestor}(v, d_1 - 1))).$$

This takes  $O(k) = O(\log n)$  time.

#### 4.2. Speeding up queries

To speed up the query, we use the linear space predecessor structure of Willard (1983) to store the  $P$  array. Furthermore, we will have a number of predecessor structures, one for each level of  $T$ , to store entries in the Euler tour  $E$ . By the property of Euler tour, the level- $d$  ancestor of a node  $v$  is the predecessor of  $i$  (where  $i$  is the principle index of  $v$ ) in the set of all indices of nodes in level  $d$  of  $T$ . Thus, both the inter-phrase and intra-phrase queries can be sped up to  $O(\log \log n)$  time while the storage complexity remains to be  $O(c(n) \log n)$ . Hence for arrays of  $k$ -bit integers, the query time is  $O(\log \log n + k)$ .

#### 4.3. Reducing storage

Although in terms of asymptotic complexity, the storage is optimal relative to the Lempel-Ziv compression, the constant factor can still be reduced. This will be important in practice. The idea is to sparsify the arrays  $P$  and  $S$  by storing only  $O(1/\log n)$ -th of the entries. Clearly, the querying for a partial phrase using the intra-phrase structure is unaffected by this change. However, some (at most  $O(\log n)$ ) of the required complete phrases are not stored in the inter-phrase structure. Fortunately, we can compute the number of 1's in those phrases by querying the inter-phrase structure. Such querying, which does not require finding a level- $d$  ancestors of a node (phrase), takes only  $O(1)$  time. Thus the storage is reduced from  $3c(n) \log n + 4c(n) + o(c(n))$  to  $c(n) \log c(n) + 6c(n) + o(c(n))$  while the query time remains to be  $O(\log n)$ .

### 5. Conclusion

We have described the first data structure that supports efficient range sum queries on a sequence of integers using space proportional to the compressed sequence when compressed using the ZL78 algorithm. We believe that our idea of combining the Lempel-Ziv compression algorithm with other succinct data structure techniques will be useful in designing other “opportunistic” data structures (i.e., structures with space proportional to the entropy of the data). In fact after our results, Sakadane and Grossi (2006) have improved our idea and designed a more general framework for designing “opportunistic data structures”. There are still more open problems in this direction. For example, can we design dynamic opportunistic data structure? Can we make use of other compression algorithms?

### References

Bender M, Farach-Colton M (2000) The LCA problem revisited. In 4th Latin American Theoretical Informatics, volume 1776 of Lecture Notes in Computer Science, pp. 88–94, Punta del Este, Uruguay, Springer-Verlag

- Burrows M, Wheeler DJ (1994) A block-sorting lossless data compression algorithms. Technical report Technical Report 124, Digital SRC Research Report
- Cover TM, Thomas JA (1991) Elements of Information Theory. Wiley Series in Telecommunications. John Wiley & Sons
- Ferragina P, Manzini G (2002) On compressing and indexing data. Technical Report TR02-02-01, Università di Pisa
- Geary R, Raman R, Raman V (2004) Succinct ordinal trees with level-ancestor queries. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1–10
- Golomb SW (1966) Run-length encodings. IEEE Transaction on Information Theory 12:399–401
- Grossi R, Vitter JS (2000) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In Proceedings of the Thirty Second Annual ACM Symposium on Theory of Computing, pp. 397–406
- Hon W-K, Sadakane K, Sung W-K (2003) Succinct data structures for searchable partial sums. In International Symp. on Algorithm and Computations, volume 2906 of Lecture Notes in Computer Science, pp. 505–516. Springer-Verlag, 2003
- Huffman DA (1952) A method for the construction of minimum-redundancy codes. In Proc. of the I.R.E. 40, pp. 1098–1101
- Poon CK (2003) Dynamic orthogonal range queries in OLAP. Theoretical Computer Science 296(3):487–510
- Raman R, Raman V, Rao SS (2001) Succinct dynamic data structures. In Workshop on Algorithms and Data Structures, volume 2125 of Lecture Notes in Computer Science, pp. 426–437
- Sadakane K (2002) Succinct representation of lcp information and improvements in the compressed suffix arrays. In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 225–232
- Sadakane K (2003) New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48:294–313
- Sadakane K, Grossi R (2006) Squeezing succinct data structures into entropy bounds. In Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms
- Willard DE (1983) Log-logarithmic worst-case range queries are possible in space  $\theta(n)$ . Information Processing Letters, 17:81–84
- Ziv J, Lempel A (1978) Compression of individual sequences by variable rate coding. IEEE Transaction on Information Theory 24(5):530–536