Rank and Select Revisited and Extended ⋆

Veli Mäkinen [1]

Department of Computer Science, P. O. Box 68 (Gustaf Hällströmin katu 2b)
FIN-00014 University of Helsinki, Finland.
vmakinen@cs.helsinki.fi

Gonzalo Navarro [2]

Center for Web Research, Department of Computer Science
University of Chile, Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

The deep connection between the Burrows-Wheeler transform (BWT) and the so-called *rank* and *select* data structures for symbol sequences is the basis of most successful approaches to compressed text indexing. Rank of a symbol at a given position equals the number of times the symbol appears in the corresponding prefix of the sequence. Select is the inverse, retrieving the positions of the symbol occurrences. It has been shown that improvements to rank/select algorithms, in combination with the BWT, turn into improved compressed text indexes.

This paper is devoted to alternative implementations and extensions of rank and select data structures. First, we show that one can use gap encoding techniques to obtain constant time rank and select queries in essentially the same space as what is achieved by the best current direct solution. Second, we extend symbol rank and select to *substring rank and select*, giving several space/time tradeoffs for the problem. An application of these queries is in *position-restricted substring searching*, where one can specify the range in the text where the search is restricted to, and only occurrences residing in that range are to be reported. In addition, arbitrary occurrences are reported in text position order. Several byproducts of our results display connections with searchable partial sums, Chazelle's two-dimensional data structures, and Grossi et al.'s wavelet trees.

⋆ Part of this work appeared in *Proc. LATIN'06, pp. 703–714, LNCS 3887.*
[1] Funded by the Academy of Finland under grant 108219.
[2] Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

1 Introduction and Related Work

Recent years have witnessed a boom in compressed text index development. A significant part of this development has been enabled by the discovery of the surprising new opportunities offered by the Burrows-Wheeler transform (BWT) [7]. One can now state the base result as follows: Take the Burrows-Wheeler transform (permutation) of a text, build so-called *rank* and *select* data structures for it, and you have a compressed index. Such index supports efficient substring queries on the text. Let us introduce some notation to display this connection more clearly.

The *indexed string matching problem* is that of, given a long text $T[1,n]$ over an alphabet Σ of size σ, build a data structure called *full-text index* on it, to solve two types of queries: (a) Given a short pattern $P[1,m]$ over Σ, *count* the occurrences of P in T; (b) *locate* those *occ* positions in T. There are several classical full-text indexes requiring $O(n \log n)$ bits of space which can answer counting queries in $O(m)$ time (such as suffix trees [2]) or $O(m + \log n)$ time (such as suffix arrays [22]). Both locate each occurrence in constant time once the counting is done.

The intense research over the last decade to reduce the space requirement of classical solutions has yielded *compressed full-text indexes* which take space proportional to that of the *compressed* text and *replace* it. The most succinct current structures require $nH_k(T) + o(n \log \sigma)$ bits of space, for any $k \le \alpha \log_\sigma n$ and constant $\alpha < 1$. Here, $H_k(T) \le \log \sigma$ denotes the k-th order empirical entropy of T [23] [3], a lower bound to the number of bits per symbol achievable by any compressor that considers contexts of length k to model T.

One of these structures [13] achieves $O(m \log \sigma)$ time for counting, while each occurrence can be located in $O(\log^{1+\varepsilon} n)$ time, for any constant $\varepsilon > 0$. This structure builds on three simple concepts: (i) *rank* queries over the BWT, (ii) wavelet trees, (iii) compression boosting. Let us go into further details.

The BWT. Ferragina and Manzini [12] discovered that $O(m)$ *rank* queries over $S = bwt(T)$, the BWT permutation of T, suffice to solve a counting query, whereas locating can be carried out by sampling the suffix array and using the BWT mechanism to locate the closest sample. Those rank queries are defined as follows: $rank_c(S,i)$ is the number of occurrences of symbol c in $S[1,i]$. Then, the problem of compressed full-text indexing boils down to the problem of solving *rank* queries in little space.

[3] In this paper log stands for $\log_2$ and we define $\log 0 = 0$.

The wavelet tree. This data structure, introduced by Grossi et al. [15,16], permits reducing *symbol rank* queries (i.e., over an alphabet of size σ) to *binary rank* queries (over a bit sequence). The wavelet tree is a perfectly balanced tree of height $\lceil \log \sigma \rceil$. Each tree node corresponds to a subinterval of $[1, \sigma]$ and represents the text subsequence of characters in that subinterval. At each node, the current alphabet range is partitioned into two halves, and the corresponding alphabet subintervals are assigned to the left and right children of the node. The only data stored at a node is a bitmap where, for each character of the text it represents, it is indicated whether that character went left or right. Each bitmap is processed for (binary) *rank* and *select* queries. The latter is the inverse of *rank*: $select_c(S, j)$ gives the position of the j-th occurrence of c in sequence S. If those queries can be solved in constant time, the wavelet tree solves in $O(\log \sigma)$ time the symbol *rank* and *select* queries, and it also obtains $S[i]$. If each of the binary sequences B are represented in $|B|H_0(B) + o(|B|)$ bits of space (as shown soon), the wavelet tree over sequence $S[1, n]$ requires $nH_0(S) + o(n \log \sigma)$ bits.

Compression boosting. This is a mechanism devised by Ferragina et al. [11] to partition $S = bwt(T)$ so that, by compressing each partition to its zero-order entropy ($H_0(S_i)$), one achieves $nH_k(T)$ overall, for any $k \leq \alpha \log_\sigma n$ and constant $\alpha < 1$.

Later improvements to solve symbol *rank* queries combined with multiary wavelet trees permit reducing the counting complexity to $O(m\lceil \log \sigma / \log \log n \rceil)$, which is $O(m)$ if $\sigma = O(\text{polylog}(n))$ [14]. Wavelet trees with binary *rank* and *select* are also essential in the construction of an alternative compressed full-text index [15,16] which, although not based on the BWT, obtains essentially the same space and time tradeoffs of [13].

We have gone into those details to display the deep connection between the BWT and *rank* and *select* data structures for compressed text indexing. We have also shown how improvements on the latter, combined with the BWT, turn into improved compressed text indexes. In this paper we focus on revisiting and extending the existing solutions for *rank* and *select* data structures, in order to give alternative solutions to known problems and also to face new problems of interest in text indexing.

Several byproducts derive from this work. We show how the solutions to binary *rank* and *select* can be adapted to the well-known *searchable partial sums* problem. We obtain an improved version of a well-known two-dimensional range search data structure by Chazelle [8], taking less space and with some extra functionality. We also show how wavelet trees are suitable for two-dimensional range searching, and their connection with Chazelle's data structure.

In the rest of this section we give more details on the state of the art and our contributions.

1.1 Rank and Select Revisited

We first study the *rank* and *select* problem on binary sequences, focusing on the case where the 1s (or alternatively the 0s) are sparse. We are given a binary sequence $B_{1,n}$, $B_i \in \{0,1\}$ for $1 \leq i \leq n$, and want to compress it while at the same time supporting several operations on it. Typical operations that are required are the following:

B_i:	Accesses the i-th element of the compressed sequence.
$rank_b(B,i)$:	Returns the number of times bit b appears in the prefix $B_{1,i}$.
$select_b(B,j)$:	Returns the position i of the j-th appearance of bit b in $B_{1,n}$.

Other useful operations are $prev_b(B,i)$ and $next_b(B,i)$, which give the position of the previous/next bit b from position i. These operations (as well as access to B_i) can be expressed via a constant number of *rank* and *select* queries, and hence are usually not considered separately. Notice also that $rank_0(B,i) = i - rank_1(B,i)$, so considering $rank_1(B,i)$ will be enough. However, the same duality does not hold for *select*, and we have to consider both $select_0(B,j)$ and $select_1(B,j)$. We call a representation of B *complete* if it supports all the listed operations in constant time. A representation is *partial* if it supports the listed operations only for 1-bits, that is, it supports $rank_b(B,i)$ only if $B_i = 1$ and it only supports $select_1(B,j)$.

The study of succinct representations of various structures, including binary sequences, was initiated by Jacobson [19]. The main motivation to study these operations came from the possibility to simulate tree traversals in small space: it is possible to represent the shape of a tree as a binary sequence, and then the traversal from a node to a child and vice versa can be expressed via constant number of *rank* and *select* operations. Currently, these queries are applied to many other problem domains. Especially, as explained, they have a significant role in *compressed full-text indexes* [17,12,30,31,15,25,16,14].

Jacobson showed that attaching a data structure of size $o(n)$ to the binary sequence $B_{1,n}$ is sufficient to support *rank* operation in constant time on a RAM machine. He also studied *select* operation, but for the RAM model the solution was not yet optimal. Later, Munro [24] and Clark [9] obtained constant time *select* on the RAM model, using $o(n)$ extra space as for *rank*.

Although the $n+o(n)$ solutions are asymptotically optimal for incompressible binary sequences, one can obtain shorter representations for compressible ones. Consider, for example, $select_1(B,i)$ on a binary sequence with $\ell = o(n/\log n)$ 1-bits. One can store all answers explicitly using $O(\ell \log n) = o(n)$ bits.

Pagh [27] was the first to study compressed representations of binary sequences supporting more than just access to B_i. He gave a representation of binary sequence $B_{1,n}$ that uses $\left\lceil \log \binom{n}{\ell} \right\rceil + o(\ell) + O(\log\log n)$ bits, where ℓ is the number of 1-bits in B. In principle this representation supported only B_i queries, yet it also supported *rank* queries for sufficiently dense binary sequences, $n = O(\ell \operatorname{polylog}(\ell))$. Notice that $nH_0(B) - O(\log n) \leq \log \binom{n}{\ell} \leq nH_0(B)$, where $H_0(B) = \ell \log \frac{n}{\ell} + (n-\ell)\log \frac{n}{n-\ell}$ is the *zeroth order entropy* of B.

This result was later enhanced by Raman et al. [29], who developed a *partial* representation with similar space complexity, $nH_0(B)+o(\ell)+O(\log\log n)$ bits, supporting *rank* and *select*. They also provide a new complete representation requiring $nH_0(B) + O(n \log\log n/\log n)$ bits.

As explained earlier, these solutions over binary sequences can be generalized to arbitrary sequences by using wavelet trees. Very recently, Sadakane and Grossi [32] developed a general technique that allows improving the above $nH_0(S)$ terms to $nH_k(S) + O(\frac{n}{\log n}((k+1)\log\sigma + \log\log n))$, for any $k \geq 0$. This technique, again, builds on binary *rank* and *select* queries.

The best current complete representation of binary sequences [29] is based on a numbering scheme. The sequence is divided into short chunks, which are expressed as a pair (l, i), where l is the number of 1-bits in the chunk and i is the identifier of that particular chunk among all chunks with l 1-bits. This way, chunks with few (or many) 1s require shorter identifiers and zero-order compression is achieved.

An alternative study, based on *gap encoding*, was initiated by Sadakane [31]. By encoding the distances between consecutive 1-bits (assuming 1s are minoritary), Sadakane showed that the space required was, essentially, $nH_0(B)$ bits for the binary sequence. Structures of $o(n)$ bits were attached to this representation to provide constant time access.

Grossi et al. considered the possibility of avoiding the $o(n)$ extra term, and depending only on nH_0, so that the space depends mostly on ℓ and only logarithmically on n. In the preliminary journal version of [16] they show that *rank* and *select* can be supported in $O(\log \ell)$ time, by attaching $o(nH_0)$-size information that permits binary searching the code. Blandford and Blelloch [6] presented a technique to simulate a given space-demanding data structure using $O(nH_0)$ bits of space and maintaining the same time complexity. Recently, Gupta et al. [18] (journal version in this same issue) improved these results for rank/select dictionaries, achieving $o(nH_0)$ bits on top of the gap encoding. Their time complexities approach the lower bound for this problem when the size of the structure depends on ℓ and only logarithmically on n.[4] Finally, some dynamic schemes building on gap encoding have been presented which permit insertions and deletion of bits [6,21], and achieve $O(\log n)$ time for all the operations.

Therefore, the current situation is that, if the extra directories on top of the gap encoding depend only logarithmically on the total number of bits n, then operations *rank* and *select* require time $\omega(1)$. Otherwise, if the directories can be of any size of the form $o(n)$, constant time should be possible, as in the solution by Raman et al. [29]. Yet, no precise formulation of this solution has yet been given. This is our contribution in this part.

In this paper we complete this picture by achieving *constant time* binary *rank* and *select* queries on top of gap encoding. Our final result is very similar to that of Raman et al. [29]: We obtain a complete representation of B taking $\alpha\ell \log \frac{n}{\ell} + O(\ell) + o(n)$ bits of space by attaching $o(n)$-size structures to the binary sequence B' that is obtained by encoding the gaps between consecutive 1s of B using *arbitrary random access self-delimiting integer codes*. Here α is a constant depending on the coding used. We achieve $\alpha = 1$ using, for example, Elias δ-encoding [10]. The price in the generality of our approach is the extra $O(\ell)$ term in our space requirement, compared to the $o(\ell)$ term of [29].

Binary sequences supporting *rank* and *select* operations have several immediate applications. With minimal adjustments they can solve problems such as: (*i*) store a sequence of ℓ increasing integers in $[1, n]$ so that one can query the amount of numbers smaller than X (rank) and locate the Y-th smallest number (select) in constant time; (*ii*) store a sequence of ℓ positive integers adding up n so that one can find the longest prefix whose summed values do not yet exceed X (rank) and compute the sum of the Y first numbers (select) in constant time; (*iii*) store a sparse set of size ℓ over an integer universe $[1, n]$ so that one can query the amount of numbers smaller than X (rank) and locate the Y-th value (select). In all these cases we require $\ell \log \frac{n}{\ell} + o(n)$ bits of space. Note that using plain representations just for the data, without any further structure to answer these queries in constant time, requires $O(\ell \log n)$ bits.

Note in particular that problem (*ii*) is a static version of the *Searchable Partial Sums* problem [28]. It can be solved in constant time using $k\ell + o(k\ell)$ bits of space, where k is the number of bits to represent the largest number. Now, since n is the sum of all the ℓ numbers, the largest number must be $\geq n/\ell$, and therefore $k \geq \log \frac{n}{\ell}$. Thus, the solution based on binary *rank* and *select*

[4] Their time complexity formula is long, but they get for example $o((\log \log n)^2)$.

is always similar or better. Moreover, both space complexities meet when all the numbers are very similar. As the numbers in the set differ more and more for fixed n, our solution improves (as $\ell \log \frac{n}{\ell}$ occurs in the worst case where all numbers are equal) and the $k\ell + o(k\ell)$-bit solution degrades (as k grows to accomodate the largest number).

1.2 Rank and Select Extended

In this paper we also introduce a new problem, *position restricted* substring searching, which consists of two new queries: (a') Given $P[1, m]$ and two integers $1 \leq l \leq r \leq n$, *count* all the occurrences of P in $T[l, r]$, and (b') *locate* those $occ_{l,r}$ occurrences. These queries are fundamental in many text search situations where one wants to search only a part of the text collection, e.g. restricting the search to a subset of dynamically chosen documents in a document database, restricting the search to only parts of a long DNA sequence, and so on. Curiously, there seem to be no solutions to this problem apart from locating all the occurrences and then filter those in the range $[l, r]$. This costs at least $O(m + occ)$ for (a') and (b') together, using classical data structures.

We present several alternative structures to solve this problem. For example, by using $O(n \log^{1+\varepsilon} n)$ bits of space, for any constant $\varepsilon > 0$, we can achieve $O(m + \log \log n)$ counting time and $O(1)$ locating time per occurrence. This worsens to $O(m + \log n)$ and $O(\log n)$ time, respectively, if we use $n \log n(1 + o(1))$ bits of space. Several of our results rely on the use of a compressed full-text index. In addition, we are able to present the occurrences in text position order, which is much more convenient than the classical suffix array order. Actually, within the same $O(\log n)$ time we are able to retrieve the k-th ocurrence, in text position order, for any given k.

Interestingly, our solutions can also be seen as extensions of *rank* and *select* queries, namely, to *substring rank and select*. For a string s, $rank_s(S, i)$ is the number of occurrences of s in $S[1, i]$, and $select_s(S, j)$ is the starting position of the j-th occurrence of s in S. As far as we know, this problem has not been addressed before. We can use the indexes for position-restricted substring searching to answer $rank_s$ in the same time of a counting query (type (a')), and $select_s$ in the same time of a counting query plus the time to locate one occurrence (type (b')).

As a byproduct, we present a more space-efficient implementation of a well-known two-dimensional range search data structure by Chazelle [8]. We show how modern *rank* and *select* data structures over bit arrays can be used to reduce the constant factor of its space requirement and to implement some extended functionalities. We also show that Grossi et al.'s wavelet trees [15,16] are suitable for two-dimensional range searching, pointing out in particular their connection with Chazelle's data structure.

Our problem falls within a more general problem studied in [5], where a set of objects with attached priorities is indexed so as to retrieve objects ordered by priority. We obtain better complexities for the particular case we address, see paragraph "Larger and faster" within Section 3.2 for details.

2 Rank and Select Revisited

2.1 Self-delimiting codes

Let us first formally define what we mean by random access self-delimiting code.

Definition 1 *Let x be an integer $x \geq 0$. A code $c(x) \in \{0,1\}^*$ is a* random access self-delimiting code *if the following conditions hold:*

a) $c(x)$ is not prefix of $c(y)$ for any integer $y \geq 0, y \neq x$;
b) $|c(x)| \leq \alpha \log x + g(x)$, where $g(x) = o(\log x)$, and α is a constant;
c) $c(x)$ can be decoded into x in constant time on the RAM model, using an auxiliary table of size $o(n)$ common to all codes, for any $x \leq n$.

An example of a random access self-delimiting code is Elias δ-code [10]:

$$\delta(x) = \underbrace{11\cdots1}_{|b(|b(x)|-1)|-1} 0b(|b(x)|-1)b(x), \tag{1}$$

where $b(x)$ is the binary representation of x. Note that x can be uniquely decoded from this representation, fulfilling property a) above, and that the representation takes $|b(x)| + 2|b(|b(x)|-1)|$ bits. As $|b(x)| + 2|b(|b(x)|-1)| \leq (\log x + 1) + 2(\log\log x + 1) = 3 + \log x + 2\log\log x$, the code fulfills property b) above with $\alpha = 1$, $g(x) = 2\log\log x + 3$. Property c) is fulfilled by noticing that a table of $2^{|b(|b(n)|-1)|+1} = O(\log n)$ entries is enough to store information of where each of the binary sequences of length $|b(|b(n)|-1)|$ contain the first 0. When $x \leq n$, we can decode it by reading three blocks of bits from $\delta(x)$.

Def. 1 also captures other Elias codes. For example, it captures γ-code:

$$\gamma(x) = \underbrace{11\cdots1}_{|b(x)|-1} 0b(x), \tag{2}$$

but the leading constant α becomes 2. On the other hand, $g(x) = 2$. There are several other codes providing trade-offs for α and $g(x)$. See e.g. [3, App. A].

Let us now examine a property of self-delimiting codes that extends property c) to short sequences of self-delimiting codes (a simpler version is used in [19] to decode Huffman codes, see also [4]).

Lemma 1 *Let* $X = c(x_1)c(x_2)\cdots c(x_p)$ *be a sequence of* $O(\log n)$ *bits representing a sequence of random access self-delimiting codes* $c(x_k)$*, where* $\sum_{k=1}^{p} x_k \leq n$*. Let* $pos(X,k) = \sum_{k'=1}^{k} |c(x_{k'})|$ *and* $dpos(X,k) = \sum_{k'=1}^{k} x_{k'}$*. Using an index of* $o(n)$ *bits, we can i) decode* $c(x_k)$ *in constant time for any given* k*; ii) compute* k*,* $pos(X,k)$*, and* $dpos(X,k)$ *such that* $pos(X,k-1) < j \leq pos(X,k)$ *in constant time for any given position* j *of* X*; and iii) compute* k*,* $pos(X,k)$*, and* $dpos(X,k)$ *such that* $dpos(X,k-1) < i \leq dpos(X,k)$ *in constant time for any given decoded position* i*. Query iii) requires the restriction* $\sum_{k=1}^{p} x_k = O(\mathrm{polylog}(n))$*.*

Proof. We partition X into a constant number of t-bit blocks, $t = \lfloor \frac{\log n}{2} \rfloor$. Let us denote one such block by x in the following. We build tables storing precomputed answers for all t-length binary sequences (blocks x) as follows. Let $G[0,\sqrt{n}-1][0,t]$ be a table such that $k' = G[x][j]$ tells the number of codes included in $x[1,j]$. Let another table $posG[0,\sqrt{n}-1][0,t]$ store the length of these codes, that is, $pos(x,k') = posG[x][k']$. Similarly, we store table $dposG[0,\sqrt{n}-1][0,t]$ such that $dpos(x,k') = dposG[x][k']$. We initialize $pos[x][0] = 0$ and $dpos[x][0] = 0$ to handle the boundary case correctly.

Note that $s = G[x][t]$ gives the number of code words in x, and then $pos(x,s) = posG[x][s]$ and $dpos(x,s) = dposG[x][s]$ give the sum of the code lengths in x and sum of the decoded values in x, respectively.

Query *i*) is handled by summing up values $G[x][t]$ into *scodes* for each consecutive t-bit block x of X, until $scodes + G[x][t] \geq k$. Then we have found the correct block x and can query $posG[x][k-1-scodes]$ to reveal where the $(k-1)$-th code ends, and finally decode the k-th code in constant time. Notice that the consecutive blocks may overlap when a suffix of a block does not contain the complete code word. Hence, we read the first t bits of X to integer x, continue reading the next t bits from position $posG[x][G[x][t]]+1$ of X to x, an so on. If some code word spans more than two blocks, we decode it in constant time using property *c*) of self-delimiting codes, and continue scanning from the end of that code word. Overall, we use time linear in the number of blocks scanned (which is constant), since only one code word per block needs special attention.

Query *ii*) is analogous by summing up instead values $posG[x][G[x][t]]$ into *spos*, until $spos + posG[x][G[x][t]] \geq j$. Then $k' = G[x][j - spos - 1]$, $pos(x,k') = posG[x][k']$, and $dpos(x,k') = dposG[x][k']$. As we sum up $posG[x][G[x][t]]$ values, we also add up values $G[x][t]$ into *scodes* and $dposG[x][G[x][t]]$ into *sdpos*. The required values are then computed in constant time as $k = scodes +$

$k'+1$, $pos(X,k) = spos + pos(x,k'+1)$, and $dpos(X,k) = sdpos + dpos(x,k'+1)$. The case were a single code word spans several blocks is easily taken into account in the computation, as in case i).

Query iii) proceeds similarly by summing up the values $dposG[x][G[x][t]]$ into $sdpos$ until $sdpos + dposG[x][G[x][t]] \geq i$. We use another precomputed table $H[0,\sqrt{n}][0, c \cdot \log^d n]$, where c and d are the constants in the $O(\mathrm{polylog}(n))$ restriction of case iii). Value $k' = H[x][i']$ gives the maximum k' such that $dpos(x,k') \leq i'$. Hence, using $i' = i - sdpos - 1$, we get $k' = H[x][i']$ and $k = scodes + k' + 1$, $pos(X,k) = spos + pos(x,k'+1)$, and $dpos(X,k) = sdpos + dpos(x,k'+1)$ are the required values to be computed, where $scodes$ and $spos$ are computed as in case ii). □

2.2 Compressing binary sequences

We compress the binary sequence $B_{1,n}$ into $B'_{1,n'}$ using self-delimiting encoding to represent the lengths of 0-runs between consecutive 1-bits: Let $X = x_0, x_1, \ldots, x_\ell$ be the sequence of integers such that $x_i = select_1(B, i+1) - select_1(B,i) - 1$, where ℓ is the number of 1-bits in B, $select_1(B,0) = 0$ and $select_1(B, \ell+1) = n+1$. That is, x_i is the length of the $(i+1)$-th 0-run. Then, $B' = c(x_0)c(x_1)\cdots c(x_\ell)$. For example, $B = 000100110100$ is encoded as $B' = c(3)c(2)c(0)c(1)c(2)$.

Before explaining how to support *rank* and *select* using B', let us analyze the size of the encoding.

Lemma 2 *Using random access self-delimiting code $c()$, a binary sequence $B_{1,n}$ can be compressed into binary sequence $B'_{1,n'}$ such that $n' \leq \alpha\ell \log\frac{n}{\ell}(1 + o(1)) + O(\ell + \log n)$, where ℓ is the number of 1-bits in B, and α is the constant in Def. 1.*

Proof. The length of B' is maximized when all the 1-bits are equally distributed in B, that is, $x_i = (n-\ell)/(\ell+1)$ for all i. Since $|c(x_i)| \leq \alpha \log x_i + o(\log x_i)$, we have $n' \leq \alpha(\ell+1)\log\frac{n-\ell}{\ell+1}(1+o(1)) + O(\ell) \leq \alpha\ell\log\frac{n}{\ell}(1+o(1)) + O(\ell + \log n)$ as claimed. This encompasses the cases $\ell = o(n)$ (i.e., $x_i = \omega(1)$) and $\ell = \Theta(n)$ (i.e., $x_i = \Theta(1)$). □

Notice that $\ell\log\frac{n}{\ell} \leq \ell\log\frac{n}{\ell} + (n-\ell)\log\frac{n}{n-\ell} = nH_0(B) = \ell\log\frac{n}{\ell} + O(\ell)$, since $(n-\ell)\log\frac{n}{n-\ell} \leq \ell/\ln 2$. We can hence re-express the size of B' as $\alpha n H_0(B)(1 + o(1)) + O(\ell + \log n)$. Recall that, using δ-encoding, we achieve $\alpha = 1$.

We need the following lemma that characterizes a non-stretching property on B'.

Lemma 3 *Let $p = select(B, i)$ and $q = select(B, j)$ for any $i < j$, and p' and q' be the positions of B' starting codes $c(x_i)$ and $c(x_j)$. Then, $q' - p' = O(q - p)$.*

Proof. We can bound $q' - p'$ similarly as for n' in the proof of Lemma 2: $q' - p' = \sum_{k=i}^{j-1} |c(x_k)| \le (j - i)\alpha \log \frac{q-p-(j-i)}{j-i}(1 + o(1)) + O(j - i) \le (j - i)\alpha \frac{q-p}{j-i} + O(j - i) \le (q - p)(\alpha + O(1))$. □

2.3 Supporting rank

We first notice that if we are given block C of length $\log n$ in B, then the corresponding block $X = c(x_0)c(x_1)\cdots c(x_p)$ of B' is of length $O(\log n)$ by Lemma 3. Here *corresponding* means the smallest block sequence that, when decoded, contains C (decoding $c(x_i)$ gives x_i 0's followed by a 1). We have the connection

$$rank_1(C, i + offset) = k, \tag{3}$$

where k is the minimum value such that $k + \sum_{k'=0}^{k} x_{k'} \ge i$ (sum of the length of 0-runs plus number of 1-bits), and *offset* tells where the block C starts inside x_0. This is almost identical to query *iii*) of Lemma 1, where the value of k is computed in constant time by maximizing $dpos(X, k) = \sum_{k'=0}^{k} x_{k'} < i$. This change to Lemma 1 is straightforward, and hence we can compute k in Eq. (3) in constant time given X and i.

To compute $rank_1(B, i)$ we store $D[i/\log n] = rank_1(B, i)$ for i multiple of $\log n$, where $D[0] = 0$.[5] That is, B is divided into blocks of length $\log n$ for which the *rank* at the start of the block can be computed by table lookup. We have

$$rank_1(B, i) = D[\lfloor i/\log n \rfloor] + rank_1(B_{j+1\ldots j+\log n}, i - j), \tag{4}$$

where $j = \lfloor i/\log n \rfloor \cdot \log n$.

Another table stores pointers to B': $D^p[i/\log n]$ gives the starting position of the block corresponding to $B_{i+1\ldots i+\log n}$ in B' for i multiple of $\log n$. Another table *offsetD*$^p[i/\log n]$ stores the offsets inside the corresponding blocks. Eq. (3) gives then the way to compute *rank* inside the block in B'. The final condition of query *iii*) holds because our blocks $C = B_{j+1\ldots j+\log n}$ are of length $\log n$ once decompressed. It is still possible that the first or last x_i value in the corresponding X is not $O(\mathrm{polylog}(n))$, but one can treat the first and last block individually, without resorting to table H, and still retain constant time.

[5] To clarify notation we assume logarithms to give integers. In general one should take floors.

Notice that tables D, D^p, and $\mathit{offsetD}^p$ require each $O((n/\log n)\log n) = O(n)$ bits, which is too much. However, we can use the standard trick [19] of storing absolute values for every $\log^2 n$-th position (superblock) and relative values for each $\log n$-th position (block): Let $D1[i/\log^2 n] = rank(B,i)$ for i multiple of $\log^2 n$, and $D2[i/\log n] = D[i/\log n] - D1[\lfloor i/\log^2 n\rfloor]$ for i multiple of $\log n$. Then $D[i/\log n] = D1[\lfloor i/\log^2 n\rfloor] + D2[i/\log n]$ for i multiple of $\log n$. Table $D1$ only requires $n/\log n$ bits, and table $D2$ requires $n\log\log n/\log n$ bits, as the maximum value in $D2$ is $\log^2 n$. Due to Lemma 3, analogous replacements can be done for table D^p. Finally, we can do the same in $\mathit{offsetD}^p$ because the difference between two consecutive table values cannot exceed $\log n$. This ensures that all tables will take at most $O(n\log\log n/\log n) = o(n)$ bits.

We have shown that the structure supports constant-time $rank$ using $\alpha\ell\log\frac{n}{\ell}(1+o(1))+O(\ell)+o(n)$ bits of space, where the first part comes from the size of B' (Lemma 2) and $o(n)$ comes from the $rank$ data structures, from the structures of Lemma 1, and from the $O(\log n)$ of Lemma 2.

2.4 Supporting select

Providing constant time $select_1(B,j)$ uses similar ideas as for $rank$, but some parts become more complicated. We explain the difficult parts in detail and sketch those analogous to the $rank$ solution.

We use tables $E1$ and $E2$ like $D1$ and $D2$ for $rank$, but this time storing every $\log^2 n$-th $select_1$ answer in $E1$ and every $\log^{1/2} n$-th relative $select_1$ answer in $E2$. More precisely, $E1[\lfloor j\log^{1/2} n/\log^2 n\rfloor] + E2[j]$ gives the position in B' where the code of $x_{1+j\log^{1/2} n}$ begins. Notice that the maximal value in $E2$ table is $O(\log^3 n)$, hence $O(\log\log n)$ bits are enough for each entry. Both tables $E1$ and $E2$ take $o(n)$ bits.

We can now find the starting positions of every $\log^{1/2} n$-th code (let us call blocks the areas of B between consecutive sampled positions). However, the distance between two sampled positions (i.e., block length) in B' can be $O(\log^{1/2} n\log n)$, if all the intermediate codes use the maximum $O(\log n)$ bits. Using the technique of Lemma 1 we could need $O(\log^{1/2} n)$ time to find the starting position of a non-sampled code. To avoid this, we separate the blocks into *small* and *large*. A block is large if its length in B' is greater than $\log n$, otherwise it is small. Notice that we can find the j-th code inside a small block in constant time using Lemma 1. For large blocks, we store all the answers (that is, corresponding code beginning following each 1-bit) explicitly. We need to show that the total number of bits used for large blocks is sub-linear: Each large block requires $O(\log^{1/2} n\log\log n)$ bits to store its answers. We can limit the amount of large blocks, say L, as follows. The sum of all values in large blocks cannot exceed n, hence L is maximized when each value is equal to n/L. Considering one block, we get inequality $\log^{1/2} n \log \frac{n}{L} > \log n$, that is, $L < n/2^{\sqrt{\log n}}$. Now the overall space needed for all the explicit answers in large blocks, that is, $O(L \log^{1/2} n \log \log n)$, can be seen to be $o(n)$.

To complete the description of $select_1$-queries, we still need to show (i) how to find the explicit answers corresponding a large block, and (ii) how to map the position in B' to a position of B (as that is the final answer we want). To solve (i) we proceed as follows. As all the explicit information for large blocks takes the same number b of bits, we concatenate all the data together and store a bitmap telling which blocks are large. Then, computing $rank_1$ over this bitmap and multiplying by b gives the position of the entries for large blocks. Using the technique of [19] this bitmap takes only $O(n/\log^{1/2} n)$ bits.

Solving (ii) is trickier, but the solution uses again the small/large blocks approach. We sample codes of B' building superblocks of length $\log^2 n$ and blocks of length $\log n$. However, even storing the relative pointers from blocks of B' to the corresponding positions in B may require $O(\log n)$ bits. To avoid this, we divide the *superblocks* into small and large; a superblock is small if every relative pointer value in its blocks takes at most $\log^{1/2} n$ bits, otherwise the superblock is called large. The space required for the block pointers inside small superblocks is clearly $o(n)$. Hence, inside small superblocks we get for each $\log n$-th position of B' the corresponding position in B by reading from tables; for other positions query *ii*) of Lemma 1 provides a constant-time solution. For large superblocks, we can use exactly the same strategy, but we need to use $O(\log n)$ bits for each block pointer. However, the amount of large superblocks, say S, can be bounded by noticing that at least one block inside a superblock corresponds to an area of length at least $2^{\log^{1/2} n}$ in B. Hence, $S 2^{\log^{1/2} n} \leq n$, and the total number of bits needed for all block pointers inside large superblocks is $S \log^2 n \leq n \log^2 n / 2^{\log^{1/2} n} = n\, 2^{2 \log \log n} / 2^{\log^{1/2} n} = o(n)$. Finally, the answers to small and large blocks are stored in separate tables, but this time we can afford to store direct links to the corresponding positions in these tables as the links are stored for superblocks.

Note that we are performing a regular sampling over B', where its codes do not start at regular positions. Each superblock/block also stores the offset in B' from its regular sample position to the beginning of the code word where the sample falls. This requires $O(\log \log n)$ bits per block and adds up to $o(n)$. Given a code word beginning, this information lets one know where is the beginning of the code word from where we must apply Lemma 1 until reaching the desired codeword (that is, up to which codeword has the preceding sample accumulated positions in B).

It is easy to extend the structure to allow $select_0$-queries as well. However, to do this we need to adjust our coding slightly: The runs of 1-bits are a problem, as such are encoded as a run of $c(0)$ codes, and precomputed $select_0$ answers may require $O(\log n)$ bits no matter which sample rate is used. As runs of zeros and ones alternate in B, we can simply code the sequence of both runs. This complicates slightly the details of how $rank_1$ and $select_1$ are implemented, as one has to take into account pointers inside runs of 1-bits, and preprocessing for a variant of Lemma 1. The computation time remains constant. While in general the structure gets smaller, in the worst case we may add $O(\ell)$ bits to Lemma 2; if all runs of 1-bits are of length 1 the original coding did not use any bits for them, now we use $|c(1)| = O(1)$ bits for each.

What we have achieved with the new coding is that every second code encodes at least one 0-bit. Now, we can use superblocks of size $\log^2 n$ and blocks of size $\log n$ to store indirectly pointers to each $\log n$-th 0-bit in B' (pointer to the code word containing the 0-bit, and offset inside it). The block length is at most $O(\log n)$, so we can find in constant time the j-th 0-bit inside each block by using the techniques of Lemma 1. We have obtained the following result.

Theorem 1 *There is a complete representation for a binary sequence $B_{1,n}$ requiring $\alpha\ell\log\frac{n}{\ell} + O(\ell) + o(n)$ bits, where ℓ is the number of 1-bits in B, and α is a constant depending on the random access self-delimiting code used.*

Note that we have simplified the space bound by removing the $o(\alpha\ell\log\frac{n}{\ell})$ term. The reason is that $\ell\log\frac{n}{\ell} = o(n)$ when $\ell = o(n)$, and otherwise the $O(\ell)$ term hides the constant $\log\frac{n}{\ell}$.

We remark that queries $prev_b(i)$ and $next_b(i)$ can be directly supported by small changes to the *rank* mechanism, without requiring the structures for *select*.

Minor modifications (simplifications) to $rank_1$ and $select_1$ structures give constant time access inside a sequence of self-delimiting codes. Hence we have a corollary that applies to the searchable partial sums problem:

Corollary 1 *A sequence of positive integers $x_1x_2\ldots x_\ell$ adding up n can be represented using $\ell\log\frac{n}{\ell} + O(\ell) + o(n)$ bits of space so that the $\sum_{i=1}^{Y} x_i$ can be computed in constant time. Moreover, one can find in constant time the maximum value j such that $\sum_{i=1}^{j-1} x_i < X$ for a given limit X. The term $O(\ell)$ can be removed from the space complexity by using representations alternative to gap encoding [29].*

3 Rank and Select Extended — Position Restricted Substring Searching

3.1 Two-Dimensional Range Searching

We describe a range search data structure to query by rectangular areas. The structure is a more succinct variant of one from Chazelle [8,20], where we have replaced the original $O(n)$-bit data structure for *rank* with newer structures performing *rank* and *select* in $n + o(n)$ bits. Given a set of points in $[1, n] \times [1, n]$, the data structure permits determining the number of points that lie in a range $[i, i'] \times [j, j']$ in time $O(\log n)$, as well as retrieving each of those points in $O(\log n)$ time in the order given by one coordinate. The structure improved can be implemented using $n \log n(1 + o(1))$ bits when no two points share the same row or column.

Structure. We describe a slightly simpler version of the original structure [8], which is sufficient for our problem (yet our improvements can be applied to the general version as well). The simplification is that our set of points come from pairing two permutations of $[1, n]$. Therefore, no two different points share their same first or second coordinates, that is, for every pair of points $(i, j) \neq (i', j')$ it holds $i \neq i'$ and $j \neq j'$. Moreover, there is a point with first coordinate i for any $1 \leq i \leq n$ and a point with second coordinate j for any $1 \leq j \leq n$.

The structure is built as follows. First, sort the points by their j coordinate. Then, form a perfect binary tree where each node handles an interval of the first coordinate i, and thus knows only the points whose first coordinate falls in the interval. The root handles the interval $[1, n]$, and the children of a node handling interval $[i, i']$ are associated to $[i, \lfloor (i+i')/2 \rfloor]$ and $[\lfloor (i+i')/2 \rfloor + 1, i']$. The leaves handle intervals for the form $[i, i]$. All those intervals will be called *tree intervals*.

Each node v contains a bitmap B_v so that $B_v[r] = 0$ iff the r-th point handled by node v (in the order given by the initial sorting by j coordinate) belongs to the left child. Each of those bitmaps B_v is preprocessed for constant-time *rank* queries using a structure that requires $O(|B_v|)$ bits (basically, they have no superblocks but just $n/\log n$ blocks taking $O(\log n)$ bits each, recall Section 2.3). We replace this *rank* structure with the more modern ones [9,24], which take only $|B_v| + o(|B_v|)$ bits and give also *select* in constant time. This brings some complications we consider soon.

Querying. We first show how to *track* a particular point (i, j) as we go down the tree. In the root, the position given by the sorting of coordinates is precisely j, because there is exactly one point with second coordinate j for any $j \in [1, n]$. Then, if $B_{root}[j] = 0$, this means that point (i, j) is in the left subtree, otherwise it is in the right subtree. In the first case, the new position of (i, j) in the left subtree is $j \leftarrow rank_0(B_{root}, j)$, which is the number of points preceding (i, j) in B_{root} which chose the left subtree. Similarly, the new position on the right subtree it is $j \leftarrow rank_1(B_{root}, j)$.

Range searching for $[i, i'] \times [j, j']$ is carried out as follows. Find in the tree the $O(\log n)$ maximal tree intervals that cover $[i, i']$. The answer is then the set of points in those intervals whose second coordinate is in $[j, j']$. Those points form an interval in the B array of each of the nodes that form the cover of $[i, i']$. However, we need to track those j and j' coordinates as we descend by the tree. Every time we descend to the left child of a node v, we update $[j, j'] \leftarrow [rank_0(B_v, j - 1) + 1, rank_0(B_v, j')]$, and similarly with $rank_1$ for a right child. When we arrive at a node whose interval is contained in $[i, i']$, the number of qualifying points is just $j' - j + 1$. Thus the whole procedure takes $O(\log n)$ time. Figure 1 shows the pseudocode.

Algorithm RangeCount($v, [i, i'], [j, j'], [ti, ti']$)
(1) **if** $j > j'$ **then return** 0;
(2) **if** $[ti, ti'] \cap [i, i'] = \emptyset$ **then return** 0;
(3) **if** $[ti, ti'] \subseteq [i, i']$ **then return** $j' - j + 1$;
(4) $tm \leftarrow \lfloor (ti + ti')/2 \rfloor$;
(5) $[j_l, j'_l] \leftarrow [rank_0(B_v, j - 1) + 1, rank_0(B_v, j')]$;
(6) $[j_r, j'_r] \leftarrow [rank_1(B_v, j - 1) + 1, rank_1(B_v, j')]$;
(7) **return** RangeCount($left(v), [i, i'], [j_l, j'_l], [ti, tm]$) +
RangeCount($right(v), [i, i'], [j_r, j'_r], [tm + 1, ti']$);

Fig. 1. Algorithm for counting the number of points in $[i, i'] \times [j, j']$ on a tree structure rooted by v with children $left(v)$ and $right(v)$. The last argument is the tree interval handled by node v. The first invocation is RangeCount($root, [i, i'], [j, j'], [1, n]$).

For retrieving the points, we basically continue the counting process even when the nodes are completely contained in $[i, i']$. We track down the occurrences until the leaves, where their i coordinate is revealed. Internal nodes with no occurrences in the range $[j, j']$ are abandoned. The process takes at most $O(\log n)$ time per retrieved element. Figure 2 gives the pseudocode.

Note that leaves are reported in the order of their i coordinate, and moreover only the i coordinate of the solutions is delivered. In order to retrieve the j coordinate of an occurrence, we must track its local j position upwards until the tree root. This is valid both for a leaf and for a given local j coordinate at an internal tree node. For example we may wish to know the j coordinates of

```
Algorithm RangeLocate(v, [j, j'], [ti, ti'])
    if j > j' then return;
    if ti = ti' then { output ti; return; }
    tm ← ⌊(ti + ti')/2⌋;
    [j_l, j'_l] ← [rank_0(B_v, j − 1) + 1, rank_0(B_v, j')];
    [j_r, j'_r] ← [rank_1(B_v, j − 1) + 1, rank_1(B_v, j')];
    RangeLocate(left(v), [j_l, j'_l], [ti, tm]);
    RangeLocate(right(v), [j_r, j'_r], [tm + 1, ti']);
```

Fig. 2. Algorithm to invoke instead of returning $j' - j + 1$ in line (3) of RangeCount, so as to locate occurrences instead of just counting them.

the results and not the i coordinates. In this case we would not track down the occurrences from the nodes where the counting finished, but we would track them up. To track a position j upwards from node v, we do as follows: If v is the left child of its parent v_p, then the position corresponding to j in v_p is $j' = select_0(B_{v_p}, j)$. If v is a right child, then $j' = select_1(B_{v_p}, j)$. When we reach the root node we have the coordinate j of the occurrence.

Space. We do not need any pointer for this tree. We only need $1 + \lceil \log n \rceil$ bit streams, one per tree level. All the bit streams at level h of the tree are concatenated into a single one, of length exactly n. A single *rank* structure is computed for each whole level, totalizing $n \log n(1 + O(\log\log n / \log n))$ bits. Maintaining the initial position p of the sequence corresponding to node v at level h is easy. There is only one sequence at the root, so $p = 1$ at level $h = 1$. Now, assume that the sequence for v starts at position p (in level h), and we move to a child (in level $h + 1$). Then the left child starts at the same position p, while the right child starts at $p + rank_0(B_v, |B_v|)$. The length of the current sequence $|B_v|$ is also easy to maintain. The root sequence is of length n. Then the left child of v is of length $rank_0(B_v, |B_v|)$ and the right child is of length $rank_1(B_v, |B_v|)$. Finally, if we know that v starts at position p and we have the whole-level sequence B^h instead of B_v, then $rank_b(B_v, j) = rank_b(B^h, p - 1 + j) - rank_b(B^h, p - 1)$.

Note that this arrangement by levels is necessary to ensure that the $o(|B_v|)$ space complexities actually add up $o(n)$ per level, which would not happen near the bottom of the tree if we indexed each vector separately. This was not a concern in Chazelle's original $O(|B_v|)$-bits scheme. The space we achieve is asymptotically optimal in the worst case, as $n \log n$ bits are stored to store a permutation.

To move upwards in the tree with this levelwise concatenation of bit vectors, we must know the starting position of the parent of a node in its levelwise bit vector. This is easily achieved by algebraic calculations. For example, if n

is a power of 2, then the j-th ancestor of the i-th leaf starts in its bit vector at position $\lfloor i/2^j \rfloor \cdot 2^j$. Still, note that all the algorithms in this section first descend to the nodes where the upward traversals start. Thus the recursion stack contains the information on the limits in the bit arrays of all the ancestors of each relevant node.

Wavelet Trees. We note now that wavelet trees have yet other applications not considered before. Assume we have a set of points $(i, j) \in [1, n] \times [1, n]$ which is the product of two permutations of $[1, n]$ as explained in the beginning of this section. Call $i(j)$ the unique i value such that (i, j) is a point in the set. Then consider the text $T[1, n] = i(1)i(2)i(3)\ldots i(n)$. Then, *the wavelet tree of T is exactly the data structure we have described in this section.* This text has alphabet of size n and its zero-order entropy is also $\log n$, thus this wavelet tree takes $n \log n(1 + o(1))$ bits as expected. This shows that the wavelet tree structure can indeed be used to solve two-dimensional range search queries in $O(\log n)$ time, and report each occurrence in $O(\log n)$ time as well.

3.2 A Simple $O(m + \log n)$ Time Solution

Let us now address the position-restricted substring search problem.

Our first solution is composed of two data structures. The first is the familiar suffix array $\mathcal{A}[1, n]$ of T, enriched with longest common prefix (lcp) information [22]. This structure needs $2n\lceil \log n \rceil$ bits and permits determining the interval $\mathcal{A}[sp, ep]$ of suffixes that start with $P[1, m]$ in $O(m + \log n)$ time [22]. The second is the range search data structure $\mathcal{R}$ described in Section 3.1, indexing the points $(\mathcal{A}[j], j)$. Both structures together require $3n \log n(1 + o(1))$ bits, or $3n + o(n)$ words.

To find the number of occurrences of $P[1, m]$ in $T[l, r]$, we first find the interval $\mathcal{A}[sp, ep]$ of the occurrences of P in T, and then count the number of points in the range $[l, r-m+1] \times [sp, ep]$ using $\mathcal{R}$. This takes overall $O(m+\log n)$ time. Additionally, each first coordinate (that is, text position $l \le i \le r - m + 1$) of an occurrence can be retrieved in $O(\log n)$ time, that is, the $occ_{l,r}$ occurrences can be located in $O(occ_{l,r} \log n)$ time.

A plus of the index is that, unlike plain suffix arrays, this structure locates the occurrences in text position order, not in suffix array order. By walking the tree upwards from an occurrence position j within a tree node, one can also reveal its suffix array location. We note that retrieving the occurrences in text position order is interesting even if we do not want position-restricted queries. A classical scheme would pay $O(m + occ \log occ)$ to report the occurrences in text position order, whereas our scheme requires $O(m + occ \log n)$. However, our technique is *online*: after $O(k \log n)$ time we have already output the first k occurrences in the text, whereas the classical scheme requires at least $O(occ + k \log occ)$ time. (If we still wish to report occurrences in suffix array order, we should rather index the points $(i, \mathcal{A}[i])$ and search for the interval $[sp, ep] \times [l, r-m+1]$.)

More ambitious than retrieving occurrences in text position order is to be able to return the k-th occurrence, for any k, in text position order. This can also be done in $O(\log n)$ time by a slight modification of the locating algorithm. We must start by running the counting query. It will give us $O(\log n)$ tree nodes that cover the interval $[l, r-m+1]$. Now, to track down the k-th of the occurrences, we first traverse the nodes linearly, adding up the total number of occurrences found within each node (this number is $j' - j + 1$ in line (3) of RangeCount). If at some node v this sum exceeds k, then the k-th occurrence is to be found within the subtree rooted at v. We go left or right depending on how many occurrences are found in the left subtree, until reaching the proper leaf. The whole process takes $O(\log n)$ time. Fig. 3 gives the pseudocode of a recursive version of the algorithm.

Algorithm kLocate$(k, v, [i, i'], [j, j'], [ti, ti'])$
if $k = 0 \vee j > j'$ **then return** k;
if $[ti, ti'] \cap [i, i'] = \emptyset$ **then return** k;
if $[ti, ti'] \subseteq [i, i']$ **then**
    if $k \le j' - j + 1$ **then** kFind$(k, v, [j, j'], [ti, ti'])$; **return** 0;
    else return $k - (j' - j + 1)$;
$tm \leftarrow \lfloor (ti + ti')/2 \rfloor$;
$[j_l, j'_l] \leftarrow [rank_0(B_v, j-1) + 1, rank_0(B_v, j')]$;
$[j_r, j'_r] \leftarrow [rank_1(B_v, j-1) + 1, rank_1(B_v, j')]$;
$k \leftarrow$ kLocate$(k, left(v), [i, i'], [j_l, j'_l], [ti, tm])$;
$k \leftarrow$ kLocate$(k, right(v), [i, i'], [j_r, j'_r], [tm+1, ti'])$;
return k;

Algorithm kFind$(k, v, [j, j'], [ti, ti'])$
if $ti = ti'$ **then** { output ti; **return**; }
$tm \leftarrow \lfloor (ti + ti')/2 \rfloor$;
$[j_l, j'_l] \leftarrow [rank_0(B_v, j-1) + 1, rank_0(B_v, j')]$;
$[j_r, j'_r] \leftarrow [rank_1(B_v, j-1) + 1, rank_1(B_v, j')]$;
if $k \le j'_l - j_l + 1$
    then kFind$(k, left(v), [j_l, j'_l], [ti, tm])$;
    else kFind$(k - (j'_l - j_l + 1), right(v), [j_r, j'_r], [tm+1, ti'])$;

Fig. 3. Algorithm kLocate is invoked like RangeCount, plus an initial k indicating which occurrence to output. It returns 0 if it could output it, and $k - occ_{l,r}$ otherwise. kFind is used internally to output the text position of the k-th occurrence.

Substring *rank* and *select*. An extension to the classical symbol *rank* and *select* problems is *substring rank* and *substring select* problems. That is, given $s \in \Sigma^*$, $rank_s(T, i)$ is the number of occurrences of s in $T[1, i]$, while $select_s(T, j)$ is the initial position of the j-th occurrence of s in T.

Those queries are particular cases of what we have obtained with the structures in this section: $rank_s(T, i)$ corresponds just to counting the occurrences of pattern s in the interval $[1, i]$ of T, whereas $select_s(T, j)$ corresponds to finding the j-th occurrence of pattern s in the interval $[1, n]$ of T.

Larger and faster. In [5], they give a very general solution to report the top-k ranked occurrences, where the rank can be defined in any way and there are few restrictions on the search structure and data set. If we understand rank as the inverse of the text position, the set as the text suffixes, and the search structure as the suffix array, their solution permits obtaining the first k occurrences, in text position order, in time $O(m + \log n + k)$. Essentially, they use Chazelle's structure to index the points $(i, \mathcal{A}[i])$, so that one can read $\mathcal{A}$ in the leaves of the tree, and each internal node contains a section of $\mathcal{A}$ with the values in text position order. Parent nodes merge the postions of their children. To solve the query they locate leaves sp and ep, and find the maximal nodes covering $[sp, ep]$ upwards. Then they use special priority queues to merge the occurrence lists of the $O(\log n)$ nodes, which obtain the first k elements in $O(k)$ time. In order to retrieve the absolute values within each node, they use a variant of Chazelle's structure [8] that requires $O(n \log^{1+\epsilon} n)$ bits of space, for any $\epsilon > 0$.

We can adapt the method to our position-restricted search problem as follows. We index the points $(i, \mathcal{A}[i])$, but search for the nodes covering $[sp, ep]$ top-down, tracking down the interval $[l, r - m + 1]$ as we move down. When we have the $O(\log n)$ maximal covering nodes v, and for each we know the local interval $[l_v, r_v]$, we merge them using the solution in [5], yet starting processing the sequence of node v from position l_v and stopping at position r_v. Overall, this solution requires $O(n \log^{1+\epsilon} n)$ bits, and can locate each position in $O(1)$ time after $O(m + \log n)$ counting time, and they are delivered in text position order.

Yet, more modern data structures let us further improve the time complexities. Instead of the structure of Section 3.1, that of Alstrup et al. [1] can be used to index the points $(\mathcal{A}[j], j)$. This structure retrieves the $occ_{l,r}$ occurrences of a range query in $O(\log \log n + occ_{l,r})$ time. In exchange, it needs $O(n \log^{1+\epsilon} n)$ bits of space, for any constant $0 < \epsilon < 1$.

Now, given the complexity $O(\log \log n)$ for the range-search part of the counting query, we could replace the suffix array by a suffix tree, so that we still have

$O(n\log^{1+\epsilon} n)$ bits of space and can solve the counting query in $O(m+\log\log n)$ time, and the locating query in constant time per occurrence. Thus, for this particular problem, we improve upon the time complexity of [5].

Smaller and slower. Alternatively, it is possible to replace the suffix array $\mathcal{A}$ and its lcp information by any of the wealth of existing compressed data structures [26]. For example, by using the LZ-index of Ferragina and Manzini [12] we obtain $n\log n(1+o(1)) + O(nH_k(T)\log^\gamma n)$ bits of space (for any $\gamma > 0$ and any $k = O(\log_\sigma \log n)$) and the same time complexities. On the other hand, we can use the alphabet-friendly FM-index of Ferragina et al. [13,14] based on the BWT to obtain $n\log n(1+o(1)) + nH_k(T)$ bits of space (for any $\sigma = o(n/\log\log n)$ and any $k \le \alpha\log_\sigma n$ for any constant $0 < \alpha < 1$). In this case the counting time raises to $O(m\lceil\log\sigma/\log\log n\rceil + \log n)$. This is still $O(m+\log n)$ if $\sigma = O(\mathrm{polylog}(n))$.

3.3 An $O(m\log\sigma)$-time Solution

We present now a solution that, given a construction parameter t, requires $nt\log\sigma(1+o(1))$ bits of space and achieves $O(m\lceil\log\sigma/\log\log n\rceil)$ time for counting the occurrences of any pattern of length $m \le t$. Likewise, each such occurrence can be located in $O(m\lceil\log\sigma/\log\log n\rceil)$ time. For example, choosing $t = \log_\sigma n$ gives a structure using $n\log n(1+o(1))$ bits of space able to search for patterns of length $m \le \log_\sigma n$. Actually, we show that this structure can be smaller for compressible texts, taking $n\sum_{k=0}^{t-1} H_k(T)$ instead of $nt\log\sigma$.

Structure. Our structure indexes the positions of all the t-grams (substrings of length t) of T. It can be thought of as an extension of the wavelet tree [15,16] to t-grams.

The structure is a perfectly balanced binary tree, which indexes the binary representation of all the t-grams of T, and searches for the binary representation of P. The binary representation $b(s)$ of a string s over an alphabet σ is obtained by expanding each character of s to the $\lceil\log\sigma\rceil$ bits necessary to code it. We index n t-grams of T, namely $b(T[1,t])$, $b(T[2,t+1])$, $\ldots$, $b(T[n,n+t-1])$. The text T is padded with $t-1$ dummy characters at the end.

The binary tree has $\ell = t\lceil\log\sigma\rceil$ levels. Each tree node v is associated a binary string $s(v)$ according to the path from the root to v. That is, $s(root) = \varepsilon$ and, if v_l and v_r are the left and right children of v, respectively, then $s(v_l) = s(v)0$

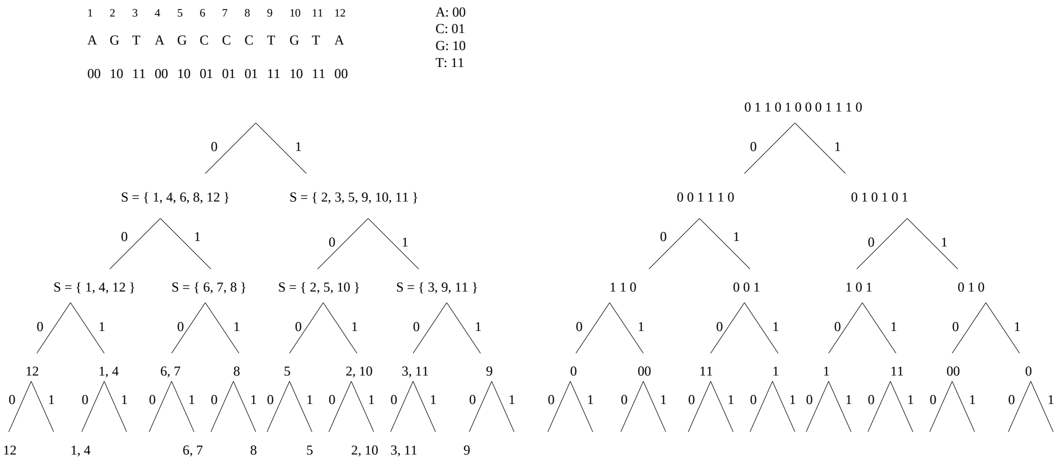

Fig. 4. An example of our structure for the text "AGTAGCCCTGTA".

and $s(v_r) = s(v)1$. To each node v we also associate a subsequence of text positions $S_v = \{i,\ s(v) \text{ is a prefix of } b(T[i, i+t-1])\}$.

Note that each $i \in S_v$ will belong exactly to one of its two children, v_l or v_r. At each internal node v we store a bitmap B_v of length $n_v = |S_v|$, such that $B_v[i] = 0$ iff $i \in S_{v_l}$. Neither $s(v)$ nor S_v are explicitly stored, only B_v is.

An example for the text "AGTAGCCCTGTA" is illustrated in Figure 4. The alphabet size is $\sigma = 4$ and we expand $t = 2$ symbols. On the left we show the text positions S that are prefixed by the binary string each node represents. On the right we display the information actually stored in the tree: a bit vector per node, telling whether the elements of its S set went to its left or right child.

Querying. Given a text position i at the root node, we can track its corresponding position in B_v for any node v such that $i \in S_v$. At the root, we start with $i_{root} = i$. When we descend to the left child v_l of a node v in the path, we set $i_{v_l} = rank_0(B_v, i)$, and if we descend to the right child v_r we set $i_{v_r} = rank_1(B_v, i)$. Then we arrive with the proper i_v value at any node v.

In order to search for P in the interval $[l, r]$, we start at the root with $l_{root} = l$ and $r_{root} = r - m + 1$, and find the tree node v such that $s(v) = b(P)$ (following the bits of $b(P)$ to choose the path from the root). At the same time we obtain the proper values l_v and r_v. Then the answer to the counting query is $r_v - l_v + 1$. The process requires $O(m \log \sigma)$ time.

To locate each such occurrence $l_v \leq i_v \leq r_v$, we must do the inverse tracking upwards, just as earlier in the paper. If v is the left child of its parent v_p, then the corresponding position in v_p is $i_{v_p} = select_0(B_{v_p}, i_v)$. If v is a right child, then $i_{v_p} = select_1(B_{v_p}, i_v)$. The final position in T is thus i_{root}. This takes

$O(m \log \sigma)$ time for each occurrence.

Space. The bulk of the space requirement corresponds to the overall size of bit arrays B_v. Vectors B_v could be represented using the technique of Clark and Munro [9,24], which provide constant-time *rank* and *select* over the bit arrays B_v using $n_v(1+o(1))$ bits. All the n_v values at any depth add up n, and since the tree height is ℓ, we have $nt\lceil \log \sigma \rceil(1+o(1))$ bits overall. The same technique used before to concatenate all the bitmaps at each level is used here to ensure that $o(1)$ is sublinear in n.

We show now that, by using a representation that achieves zero-order entropy size, the space requirement may be reduced on compressible texts T. We choose the structure of Raman et al. [29], which requires $n_v H_0(B_v) + o(n_v)$ bits to provide constant-time *rank* and *select* queries over B_v. (Note that the $O(\ell)$ overhead of the structures in Section 2 could be significant here, so we stick to the most space-efficient representation.) As we already know from the previous paragraph that the $o(n_v)$ parts add up $o(nt \log \sigma)$ bits (more precisely, $O(nt \log \sigma \log\log n / \log n)$ bits), we focus on the entropy-related part. Let us assume for simplicity that σ is a power of 2.

Let us analyze all the $n_v H_0(B_v)$ terms together. For a binary string s, let us define $n_s = |\{i,\ s \text{ is a prefix of } b(T[i, i+t-1])\}|$. Thus, if we consider vector B_{root}, its representation takes $nH_0(B_{root}) = -n_0 \log \frac{n_0}{n} - n_1 \log \frac{n_1}{n}$.

Consider now the vectors B for the two children of the root. The entropy part of their representations add up $-n_{00} \log \frac{n_{00}}{n_0} - n_{01} \log \frac{n_{01}}{n_0} - n_{10} \log \frac{n_{10}}{n_1} - n_{11} \log \frac{n_{11}}{n_1}$. We notice that $n_0 = n_{00} + n_{01}$ and $n_1 = n_{10} + n_{11}$. By adding up the size of representations of the root and its two children, we get $-n_{00} \log \frac{n_{00}}{n} - n_{01} \log \frac{n_{01}}{n} - n_{10} \log \frac{n_{10}}{n} - n_{11} \log \frac{n_{11}}{n}$ bits. This can be extended inductively to $\log \sigma$ levels, so that the sum of all the representations from the root to level $\log \sigma - 1$ is

$$- \sum_{s \in \{0,1\}^{\log \sigma}} n_s \log \frac{n_s}{n} \quad = \quad nH_0(T),$$

where $0 \log 0 = 0$.

Similarly, starting from each node v such that $s(v) \in \{0,1\}^{\log \sigma}$, we have that $nH_0(B_v) = -n_{s(v)0} \log \frac{n_{s(v)0}}{n_{s(v)}} - n_{s(v)1} \log \frac{n_{s(v)1}}{n_{s(v)}}$, and all the B vectors in the next $\log \sigma$ levels of its subtree add up

$$- \sum_{s \in \{0,1\}^{\log \sigma}} n_{s(v)s} \log \frac{n_{s(v)s}}{n_{s(v)}} \ .$$

Summing this for all the nodes representing all the possible $s(v) \in \{0,1\}^{\log \sigma}$, we have

$$-\sum_{s,s'\in\{0,1\}^{\log\sigma}} n_{ss'} \log\frac{n_{ss'}}{n_s} \quad = \quad nH_1(T).$$

This can be continued inductively until level $t\log\sigma$, to show that the overall space is

$$n\sum_{k=0}^{t-1} H_k(T) \quad + \quad O(nt\log\sigma\log\log n/\log n)$$

bits. For incompressible texts this is $nt\log\sigma(1+o(1))$, but for compressible texts it may be significantly less.

Higher arity trees. A generalization of the rank/select data structures [29] permit handling sequences with alphabets of size up to $O(\text{polylog}(n))$ with constant time $rank_c$ and $select_c$ [14]. Instead of handling one bit of $b(T[i,i+t-1])$ at a time, we could handle a bits at a time. This way, our binary tree would be 2^a-ary instead of binary. Instead of a sequence of bits B_v at each node, we would store a sequence B_v of integers in $[0,a-1]$. As long as $2^a = O(\text{polylog}(n))$ (that is, $a = O(\log\log n)$), we can index those sequences B_v with the generalized data structure so as to answer in constant time the rank/select queries we need to navigate the tree.

The search algorithm is adapted in the obvious way. When going down to the d-th child of node v, $0 \le d < a$, we update i_v to $i_{v_d} = rank_d(B_v, i_v)$ and, similarly, when going up to v from child d, $i_v = select_d(B_v, i_{v_d})$. Note that a must divide $\log\sigma$ to ensure that any pattern search will arrive exactly at a tree node. The overall time is $O(m\log(\sigma)/a) = O(m\lceil\log\sigma/\log\log n\rceil)$, either for counting or for locating an occurrence. This is $O(m)$ whenever $\sigma = O(\text{polylog}(n))$.

We note that it is necessary, again, to concatenate all sequences at each tree level, so that the limit $a = O(\log\log n)$ remains constant as we descend in the tree. For space occupancy related to entropy, the analysis is very similar; we just consider a bits at once.

Compared to the solution of Section 3.2 requiring $O(n\log n)$ bits of space and $O(m+\log n)$ counting time, we can use $t = O(\log_\sigma n)$ to achieve the same space complexity, so that any query of length up to t can be answered. The structure of this section is faster than that of Section 3.2 in this range of m values. Compared to the faster structure requiring $O(n\log^{1+\epsilon} n)$ bits and $O(m)$ counting time, this structure could answer in the same space counting queries on patterns of length up to $O(\log_\sigma n\log^\epsilon n)$. The time for counting is better than the previous structure for $m = O(\log\log n)$.

Substring rank and select Substring rank can be solved again by resorting to counting. Substring select requires more care. We search for s in the tree starting with range $[l, r] = [1, n]$. We end up at some node v (such that $s(v) = b(s)$) with $[l_v, r_v]$. To solve $select_s(T, j)$ we take entry $l_v + j - 1$ at node v and walk the tree upwards until finding the position in the root node, and that position is the answer. This takes overall time $O(|s|\lceil \log \sigma / \log \log n \rceil)$ (just as for $rank_s$), and requires $O(n|s| \log \sigma)$ bits of space (or less if T is compressible).

4 Conclusions

We have addressed several important generalizations of well-studied problems in string matching and succinct data structures. In particular, those problems find applications to compressed text indexing when combined with the Burrows-Wheeler transform.

First, we gave a new implementation of *rank* and *select* queries over sparse bit arrays based on gap encoding. This new representation obtains almost the same bounds as the best known structure. We also show a connection to the searchable partial sums problem.

Second, we generalized *rank* and *select* queries on sequences to substring *rank* and *select*, where the occurrences of any substring s can be tracked instead of only characters. Our time complexities are slightly over the ideal $O(|s|)$. These extended queries turned out to be particular cases of the more powerful *position-restricted searching*, where the search can be done inside any text substring. We have obtained space and time complexities close to those obtained for the basic problem, and moreover, we have shown that arbitrary occurrences can be delivered in text position order.

In addition, we have shown some interesting connections between well-known two-dimensional range search data structures by Chazelle and recent data structures for compressed text indexing (the wavelet trees by Grossi et al.). We also showed how modern *rank* queries permit implementing Chazelle's structure using less space and adding some extra functionality to it.

Some interesting open questions are (1) whether we can answer position-restricted counting queries in $O(m)$ time and locating each result in $O(1)$ time with structures taking $O(n \log n)$ bits of space, or even better, compressed data structures requiring $O(nH_k)$ bits of space; and (2) whether we can answer *rank* and *select* queries for a substring s in $O(|s|)$ time.

References

[1] S. Alstrup, G. Brodal, and T. Rahue. New data structures for orthogonal range searching. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

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

[3] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[4] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.

[5] I. Bialynicka-Birula and R. Grossi. Rank-sensitive data structures. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 3772, pages 79–90, 2005.

[6] D. Blandford and G. Blelloch. Compact representations of ordered sets. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[7] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

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

[9] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[10] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–20, 1975.

[11] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *OJO*, 2005.

[12] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[13] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 3246, pages 150–160, 2004.

[14] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Also as TR 2004-05, Technische Fakultät, Universität Bielefeld, Germany, December 2004.

[15] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[16] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[17] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[18] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed data structures: dictionaries and data-aware measures. In *Proc. 5th International Workshop on Experimental Algorithms (WEA)*, pages 158–169, 2006.

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

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

[21] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proc. CPM'06*, LNCS 4009, pages 306–317, 2006.

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

[23] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

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

[25] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[26] G. Navarro and V. Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, Department of Computer Science, University of Chile, April 2006. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`. Submitted to a journal.

[27] R. Pagh. Low redundancy in dictionaries with o(1) worst case lookup time. In *Proc. ICALP'99*, pages 595–604, 1999.

[28] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS'01*, pages 426–437, 2001.

[29] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 233–242, 2002.

[30] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA'02*, pages 225–232, 2002.

[31] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[32] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239, 2006.