Position-Restricted Substring Searching

Veli Mäkinen[1] ⋆ and Gonzalo Navarro[2] ⋆⋆

[1] Department of Computer Science,
University of Helsinki, Finland.
vmakinen@cs.helsinki.fi
[2] Center for Web Research
Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. A full-text index is a data structure built over a text string $T[1,n]$. The most basic functionality provided is (a) counting how many times a pattern string $P[1,m]$ appears in T and (b) locating all those occ positions. There exist several indexes that solve (a) in $O(m)$ time and (b) in $O(occ)$ time. In this paper we propose two new queries, (c) counting how many times $P[1,m]$ appears in $T[l,r]$ and (d) locating all those $occ_{l,r}$ positions. These can be solved using (a) and (b) but this requires $O(occ)$ time. We present two solutions to (c) and (d) in this paper. The first is an index that requires $O(n \log n)$ bits of space and answers (c) in $O(m + \log n)$ time and (d) in $O(\log n)$ time per occurrence (that is, $O(occ_{l,r} \log n)$ time overall). A variant of the first solution answers (c) in $O(m + \log \log n)$ time and (d) in constant time per occurrence, but requires $O(n \log^{1+\epsilon} n)$ bits of space for any constant $\epsilon > 0$. The second solution requires $O(nm \log \sigma)$ bits of space, solving (c) in $O(m\lceil \log \sigma / \log \log n \rceil)$ time and (d) in $O(m\lceil \log \sigma / \log \log n \rceil)$ time per occurrence, where σ is the alphabet size. This second structure takes less space when the text is compressible.

Our solutions can be seen as a generalization of *rank* and *select* dictionaries, which allow computing how many times a given character c appears in a prefix $T[1,i]$ and also locate the i-th occurrence of c in T. Our solution to (c) extends character *rank* queries to *substring rank* queries, and our solution to (d) extends character *select* to *substring select* queries.

As a byproduct, we show how *rank* queries can be used to implement fractional cascading in little space, so as to obtain an alternative implementation of a well-known two-dimensional range search data structure by Chazelle. We also show how Grossi et al.'s *wavelet trees* are suitable for two-dimensional range searching, and their connection with Chazelle's data structure.

1 Introduction and Related Work

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

⋆ Funded by the Academy of Finland under grant 108219.

⋆⋆ Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

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 (like suffix trees [2]) or $O(m+\log n)$ time (like suffix arrays [14]). Both locate each occurrence in constant time once the counting is done. Similar complexities are obtained with modern compressed data structures [5, 10, 7], requiring space $nH_k(T) + o(n \log \sigma)$ bits, where $H_k(T) \leq \log \sigma$ is the k-th order empirical entropy of T.[3]

In this paper we introduce a new problem, *position restricted* substring searching, which consists of two new queries: (c) 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 (d) *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 (c) and (d) together.

We present several alternative structures to solve this problem. The best complexities are summarized in Table 1.

Section	Bits of space	Counting time	Locating time
4	$O(n \log^{1+\epsilon} n)$	$O(m + \log \log n)$	$O(1)$
4	$n \log n(1 + o(1)) + O(nH_k(T) \log^{\gamma} n)$	$O(m + \log n)$	$O(\log n)$
4	$n \log n(1 + o(1)) + nH_k(T)$	$O(m\lceil \frac{\log \sigma}{\log \log n} \rceil + \log n)$	$O(\log n)$
5	$n \sum_{k=0}^{m-1} H_k(T)$	$O(m\lceil \frac{\log \sigma}{\log \log n} \rceil)$	$O(m\lceil \frac{\log \sigma}{\log \log n} \rceil)$

Table 1. Simplified complexities achieved for queries (c) and (d). Locating time is given per occurrence.

Interestingly, our solutions are also useful to solve a generalization of another well-studied problem. Given a sequence S over an alphabet Σ of size σ and a character $c \in \Sigma$, query $rank_c(S, i)$ returns the number of occurrences of c in $S[1, i]$, while $select_c(S, j)$ returns the position of the j-th occurrence of c in S. Both queries can be answered in constant time using data structures that require $nH_0(S)+o(n)$ bits of space if the alphabet of the sequence is $\sigma = O(\text{polylog}(n))$, or in $O(\log \sigma / \log \log n)$ time in general [9, 8]. They can also be solved in $O(\log \sigma)$ time using wavelet trees [10, 11]. For the case of binary sequences, apart from the simple $n + o(n)$ bits data structures [12, 4, 16], there are others that answer *rank* and *select* in constant time using $nH_0(S) + o(n)$ bits [18].

A natural generalization of the above problem is *substring rank and select*. For a string s, $rank_s(S, i)$ is the number of occurrences of s in $S[1, i]$, and $select_s(S, j)$

[3] In this paper log stands for $\log_2$.

is the starting position of the j-th occurrence of s in S. We can use the indexes for position-restricted substring searching to answer $rank_s$ in the same time of a counting query (type (c)), and $select_s$ in the same time of a counting query plus the time to locate one occurrence (type (d)).

As a byproduct, we present a space-efficient implementation of a well-known two-dimensional range search data structure by Chazelle [3]. We show in particular how the fractional cascading information (which is simulated rather than stored in Chazelle's data structure) can be represented by constant-time *rank* queries on bit arrays. We also show that Grossi et al.'s wavelet trees [10, 11] are suitable for two-dimensional range searching, pointing out in particular their connection with Chazelle's data structure.

2 Two-Dimensional Range Searching

In this section we describe a range search data structure to query by rectangular areas. The structure is a succinct variant of one from Chazelle [3, 13] where we have completely removed binary searching and fractional cascading and have replaced them by constant-time *rank* queries over bit arrays. 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. The structure can be implemented using $n \log n(1 + o(1))$ bits.

Structure. We describe a slightly simpler version of the original structure [3], which is sufficient for our problem. 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 [12, 4, 16]). The bitmaps with rank functionality give a space-efficient way to implement fractional cascading, and also avoid any need of binary searching.

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 nodes $left(v)$ 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 start from each of the tree nodes that cover $[i, i']$. Each point in the node whose second coordinate is in $[j, j']$ is tracked down in the tree until the leaves, so as to find its first coordinate i. This can be done in $O(\log n)$ time per retrieved element. (For our application, we do not describe how to associate the proper j value to each i coordinate found, but it can be done by traversing the tree upwards from each leaf using *select*.) We traverse the whole subtree of each node included in $[i, i']$, as long as it has some point in $[j, j']$. The leaves found in this process are reported. Figure 2 gives the pseudocode.

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

Algorithm RangeLocate($v, [j, j'], [ti, ti']$)
(1) **if** $ti = ti'$ **then** { output ti; **return**; }
(2) **if** $j > j'$ **then return**;
(3) $tm \leftarrow \lfloor (ti + ti')/2 \rfloor$;
(4) $[j_l, j_l'] \leftarrow [rank_0(B_v, j-1)+1, rank_0(B_v, j')]$;
(5) $[j_r, j_r'] \leftarrow [rank_1(B_v, j-1)+1, rank_1(B_v, j')]$;
(6) RangeLocate($left(v), [j_l, j_l'], [ti, tm]$);
(7) 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.

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 $rank_b(B_v, j) = rank_b(B^h, p - 1 + j) - rank_b(B^h, p - 1)$. Figure 3 shows again the counting algorithm, this time over the real data structure.

Algorithm RangeCount($B, [i, i'], [j, j'], h, p, \ell, [ti, ti']$)
(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) $[j_l, j_l'] \leftarrow [rank_0(B^h, p, p-1+j-1)+1, rank_0(B^h, p, p-1+j')]$;
(6) $[j_r, j_r'] \leftarrow [rank_1(B^h, p, p-1+j-1)+1, rank_1(B^h, p, p-1+j')]$;
(7) $[\ell_l, \ell_r] \leftarrow [rank_0(B^h, p, p-1+\ell), rank_1(B^h, p, p-1+\ell)]$
(8) $p' \leftarrow p + rank_0(B^h, \ell)$
(9) **return** RangeCount($B, [i, i'], [j_l, j_l'], h+1, p, \ell_l, [ti, tm]$) +
RangeCount($B, [i, i'], [j_r, j_r'], h+1, p', \ell_r, [tm+1, ti']$);

Fig. 3. Algorithm for counting the number of points in $[i, i'] \times [j, j']$ on the real, level-wise, structure. The first invocation is RangeCount($B, [i, i'], [j, j'], 1, 1, n, [1, n]$). We use $rank_b(B^h, a, b)$ as shorthand for $rank_b(B^h, b) - rank_b(B^h, a - 1)$.

Wavelet Trees. Wavelet trees [10, 11] are data structures for text indexing introduced by Grossi et al. 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 child 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 *rank* and *select* queries. If one uses basic techniques for those queries [12, 4, 16] the wavelet tree takes $n\lceil \log \sigma \rceil + O(n \log \log n / \log_\sigma n)$ bits of space for a text $T[1, n]$, that is, the same text size. With more advanced techniques [18], the size of the wavelet tree achieves $nH_0(T) + O(n \log \log n / \log_\sigma n)$ bits of space, where $H_0(T)$ is the zero-order entropy of T. In both cases, the wavelet tree solves in $O(\log \sigma)$ time the following queries: (a) $T[i]$, that is, finding the i-th character of T; (b) $rank_c(T, i)$, that is, finding the number of occurrences of c in $T[1, i]$; and (c) $select_c(T, j)$, that is, finding the position in T of the j-th occurrence of c.

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 $T[1, n] = i(1)i(2)i(3) \dots i(n)$. 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. Although the original wavelet-tree queries are not especially interesting in this range search scenario, we have shown in this section 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 A Simple $O(m + \log n)$ Time Solution

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 [14]. 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 [14]. The second is the range search data structure $\mathcal{R}$ described in Section 2, indexing the points $(i, \mathcal{A}[i])$. 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. In order to find them in suffix array order, we should rather index points $(\mathcal{A}[i], i)$ and search for the interval $[sp, ep] \times [l, r-m+1]$. Then $\mathcal{R}$ would retrieve the suffix array positions i (in increasing order in $\mathcal{A}$) such that $\mathcal{A}[i]$ is an occurrence.

Larger and faster. It is possible to improve the locating time to $O(1)$ by using slightly more space. Instead of the structure of Section 2, that of Alstrup et al. [1] can be used to index the points $(i, \mathcal{A}[i])$. 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 $0 < \epsilon < 1$. Thus, by using slightly more space, we achieve $O(m + \log n)$ counting time and $O(1)$ locating time per occurrence.

Given the complexity $O(\log\log n)$ for the range-search part of the counting query, it makes sense to 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.

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 [17]. For example, by using the LZ-index of Ferragina and Manzini [6] 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. [7, 8] 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 $\sigma = O(\mathrm{polylog}(n))$.

4 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 $nt \log\sigma$, where $H_k(T)$ is the k-th order empirical entropy of T [15, 10]. This is a lower bound to the number of bits per character achievable by any compressor that considers contexts of length k to model T.

Structure. Our structure indexes the positions of all the t-grams (substrings of length t) of T. It can be tought of as an extension of the wavelet tree [10, 11] 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, $s(\mathit{root}) = \varepsilon$ and, if v_l and v_r are the left and right children of v, respectively, then $s(v_l) = s(v)0$ 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.

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 $i_{v_l} = \mathit{rank}_0(B_v, i)$, and if we descend to the right child v_r we set $i_{v_r} = \mathit{rank}_1(B_v, i)$. 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 $l_v \le i_v \le r_v$, we must do the inverse tracking upwards. If v is the left child of its parent v_p, then the corresponding position in v_p is $i_{v_p} = \mathit{select}_0(B_{v_p}, i_v)$. If v is a right child, then $i_{v_p} = \mathit{select}_1(B_{v_p}, i_v)$. 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 [4, 16], which permits answering *rank* and *select* queries in constant time 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 more sophisticated techniques [18], the space requirement may be reduced on compressible texts T. In that work they represent bit array B_v using $n_v H_0(B_v) + o(n_v)$ bits, and answer *rank* and *select* queries in constant time. As we already know that the $o(n_v)$ parts add up $o(nm \log \sigma)$ bits (more precisely, $O(nm \log \sigma \log \log n / \log n)$ 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

$$- \sum_{s \in \{0,1\}^{\log \sigma}} n_s \log \frac{n_s}{n} \quad = \quad n H_0(T)$$

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 [18] permit handling sequences with alphabets of size up to $O(\text{polylog}(n))$ with constant time $rank_c$ and $select_c$ [9, 8]. 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, $0 \leq d < a$, 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 the previous section 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 the previous section in this range of m values. Compared to the faster structure requiring $O(n \log^{1+\epsilon} n)$ bits and $O(m)$ counting time, our 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 $m = O(\log \log n)$.

5 Substring Rank and Select

The techniques developed for the problem of counting and locating the occurrences of a pattern P in $T[l, r]$ can be used to solve the *substring rank* and *substring select* problem. As far as we know, this problem has not been addressed before. It extends the $rank_c$ and $select_c$ queries, $c \in \Sigma$, to strings over Σ. 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.

Note that $rank_s(T, i)$ is just the number of occurrences of s in $T[1, i]$, and therefore it corresponds directly to a particular case of our counting queries. On the other hand, $select_s(T, j)$ is solved by using the locating mechanism. We detail this query now.

With the structure of Section 3 we must start with a counting query for s in the interval $[1, n]$. Therefore, we end up at the unique interval $[sp, ep]$ at the tree root. Then, to solve $select_s(T, j)$ we must track down in the tree the position $sp + j - 1$ at the tree root. Therefore, we need overall $O(|s| + \log n)$ time for $select_s(T, j)$ (just as for $rank_s(T, i)$), yet ℓ calls to $select_s$ cost $O(|s| + \ell \log n)$. It is not clear whether the more complicated $O(n \log^{1+\epsilon} n)$ bits structure can extract random occurrences in constant time.

Let us now consider the structure of Section 4. Once 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).

6 A Small Experiment

We have implemented the simplest mechanism described in Section 3, and compared it against a brute-force solution, that is, use the plain suffix array to discover the *occ* occurrences and then pass over those determining which are in the right text range.

As the suffix array search is identical in both cases, we have focused on the time to complete the process once the suffix array range is known. For counting, the brute-force method has complexity $O(occ)$, whereas our method in Section 3 requires $O(\log n)$ time. For locating the occurrences, the brute-force method is still $O(occ)$ time, while our method requires $O(occ_{l,r} \log n)$.

We tested over different English texts, ranging from 1 to 100 megabytes. We randomly generated subintervals of the suffix array of a fixed length and compared the time to pass over it counting/reporting the text positions within some range, against using the generated suffix array interval as a key for the two-dimensional search of our method. Note that the fact that the suffix array ranges generated do not come from an actual search is irrelevant for the performance of the process, and it permits us tight control over *occ*.

According to the experiments, our counting method becomes faster than brute force approximately for $occ > 1,000$. This did not depend significatively on the text size nor on the text interval $[l, r]$ chosen. The two-dimensional search part takes, in our method, time similar to the suffix array search.

For locating queries, on the other hand, our method was superior for $\frac{occ_{l,r}}{occ} < 0.004$. This is obtained when *occ* exceeds 1,000 by a sufficient margin (say, 10 times). The reason is that our method has a constant overhead that is independent of $occ_{l,r}$, so that even for $occ_{l,r} = 0$ the brute force method is faster for $occ < 1,000$.

The ranges of values obtained show that our method is reasonably practical, and it wins when the query is sufficiently selective, as expected.

7 Conclusions

We have addressed several important generalizations of well-studied problems in string matching and succinct data structures. First, we generalized the indexed string matching problem to *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. 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 $O(|s|)$.

It is an interesting open question whether we can close those small gaps, that is (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 substring s in $O(|s|)$ time.

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 *rank* queries permit implement Chazelle's structure without using any fractional cascading information nor binary searches.

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. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
4. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.
5. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.
6. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
7. 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.
8. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. Technical Report 2004-05, Technische Fakultät, Universität Bielefeld, Germany, December 2004. Submitted to a journal.
9. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Department of Computer Science, University of Chile, Chile, August 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.
10. 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.
11. 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.
12. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS'89)*, pages 549–554, 1989.
13. J. Kärkkäinen. *Repetition-based text indexes.* PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999. Also available as Report A-1999-4, Series A.
14. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
15. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
16. I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, LNCS 1180, pages 37–42, 1996.
17. G. Navarro and V. Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2005-7, Department of Computer Science, University of Chile, Chile, June 2005. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr.ps.gz`. Submitted to a journal.
18. 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.