Succinct Representation of Sequences*

Paolo Ferragina[1], Giovanni Manzini[2], Veli Mäkinen[3], and Gonzalo Navarro[4]

[1] Dipartimento di Informatica, University of Pisa, Italy.
[2] Dipartimento di Informatica, University of Piemonte Orientale, Italy.
[3] Department of Computer Science, University of Helsinki, Finland.
[4] Center for Web Research, Department of Computer Science, University of Chile, Chile.

Abstract. Given a sequence $S = s_1 s_2 \dots s_n$ such that $1 \leq s_q \leq r$ for all q, where $r = O(\text{polylog}(n))$, we show how S can be represented using $nH_0(S) + o(n)$ bits (where $H_0(S)$ is the zero-order entropy of S), so that we can know any s_q, as well as answer *rank* and *select* queries on S, in constant time. This extends previous results on binary sequences, and improves previous results on general sequences where those queries are answered in $O(\log r)$ time. Furthermore, we show how our technique can be applied to improve a succinct full-text index.

1 Introduction

Recent years have witnessed an increasing interest on *succinct* data structures. Their aim is to represent the data using as little space as possible, yet efficiently answering queries on the represented data. Several results exist on the representation of sequences [5, 6, 1, 9, 10], trees [7, 3], graphs [7], permutations [8], etc. The structures differ depending on the types of queries supported.

A heavily studied case is that of binary sequences, with *rank* and *select* queries. Given a sequence $S = s_1 s_2 \dots s_n$, with each $s_q \in \{0, 1\}$, $\mathsf{Rank}_c(S, q)$ is the number of times symbol $c \in \{0, 1\}$ appears in $S[1, q] = s_1 s_2 \dots s_q$, and $\mathsf{Select}_c(S, q)$ is the position in S of the q-th occurrence of symbol $c \in \{0, 1\}$.

The first results on binary sequences [5, 6, 1] achieved constant time on those queries by using $n + o(n)$ bits. In those schemes, n bits are used by S itself and $o(n)$ additional bits are needed by the data structures used to answer Rank_c and Select_c queries. Further refinements [9, 10] achieved constant time on the same queries by using $nH_0(S) + o(n)$ bits, where

* Partially supported by the Italian MIUR projects `ALINWEB`, `ECD`, `Grid.it` and ``Piattaforma distribuita ad alte prestazioni'', and by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

$H_k(S)$ is the k-th order entropy of S. In this case, a further nontrivial query that is solved is determining s_q given q.

The case of general sequences, where $1 \leq s_q \leq r$ for some r, has received less attention. The only existing proposal is the wavelet tree [4], an elegant data structure that allows to reduce rank/select operations over arbitrary-alphabet sequences onto rank/select operations over binary sequences. By using results on binary sequence representations [9, 10] the wavelet tree achieves a representation using $nH_0(S) + o(n)$ bits that answers s_q and $\mathsf{Rank}_c(S, q)$ queries in $O(\log r)$ time.

In this paper we generalize a result on binary sequences [9, 10] to sequences of symbols in the range $[1, r]$, useful for small r. The main challenge in this generalization is to generate short descriptions of pieces of the sequence, which can be computed in constant time and are used to index into tables with partial precomputed queries. This is more complex than for binary sequences. We obtain a sequence representation using $nH_0(S) + O((rn \log\log n)/\log_r n)$ bits which answers queries s_q, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ in constant time. This is interesting only for small $r = o(\log n/\log\log n)$.

This base result permits us generalizing wavelet trees to be r-ary, for small r. Wavelet trees are binary in part because they need to represent a sequence telling which branch was chosen by each element of the sequence. Our generalization, in turn, strenghtens the base result since it achieves $nH_0(S) + o(n)$ bits of space and constant query time for any $r = O(\text{polylog}(n))$. That is, the same queries answered in logarithmic time with binary wavelet trees [4] are now answered in constant time.

Finally, we show how this result can be used for text indexing. A *full-text self-index* is a succinct data structure that represents a text string $T[1, n]$ (which is a sequence over an alphabet Σ) while supporting not only access to any character $T[i]$ but also the efficient search for an arbitrary pattern as a *substring* of the indexed text. It has been recently shown [2] that a full-text self-index can be represented using $nH_k(T) + o(n)$ bits of space, for any $k \leq \alpha \log_{|\Sigma|} n$, $0 < \alpha < 1$, so as to answer the following queries: (i) number of occurrences of a string pattern $P[1, p]$ in T in time $O(p \log |\Sigma|)$, (ii) initial text position of each such occurrence in time $O(\log |\Sigma| \log^{1+\varepsilon} n)$ for any $\varepsilon > 0$, and (iii) content of $T[i, i + \ell - 1]$ in time $O(\log |\Sigma|(\ell + \log^{1+\varepsilon} n))$. This result makes use of wavelet trees for its internal workings. By using our new sequence representation instead of the wavelet tree, we achieve the same space usage and the $\log |\Sigma|$ term disappears from all time complexities. The result is the fastest among

those full-text indexes that use minimum space, $nH_k(T) + o(n)$ bits (the only other indexes in this category are [4, 2]).

2 Representing sequences of small numbers

Let S be a sequence of n numbers in the range $[1, r]$, called *symbols* from now on. Our aim is to represent S essentially in the space required by its zero-order entropy, $nH_0(S)$, so as to answer $S[q]$, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in constant time. Our only assumption on r is $2 \leq r \leq \sqrt{n}$, albeit our final result will be interesting only for $r = o(\log n / \log\log n)$. We first concentrate on $S[q]$ and $\mathsf{Rank}_c(S, q)$ queries and address $\mathsf{Select}_c(S, q)$ later.

Structure. We divide S into blocks of size $u = \left\lfloor \frac{1}{2} \log_r n \right\rfloor$. We define the following sequences (which are *not* stored explicitly) of values indexed by block number $i \in 1 \ldots \lceil n/u \rceil$:

- $S_i = S[u(i-1)+1 \ldots ui]$ is the sequence of symbols of block i.
- For each symbol $c \in [1, r]$, $N_i^c = \mathsf{Rank}_c(S_i, u)$ is the number of occurrences of c in S_i.
- $L_i = \left\lceil \log \binom{u}{N_i^1, \ldots, N_i^r} \right\rceil$ is the number of bits necessary to code all possible sequences of u symbols in $[1, r]$ having N_i^c occurrences of symbol c, for $c \in [1, r]$.

We store the following information:

- For each block i, an L_i-bits identifier for the sequence S_i among those with N_i^c occurrences of symbol c, $c \in [1, r]$. The identifier is actually an index inside a table $E_{N_i^1, \ldots, N_i^r}$, which contains the actual sequences (see next). These L_i-bits identifiers are called I_i and they are of variable length. We store all them concatenated into a single binary sequence.
- For each block i, an identifier for the tuple $(N_i^1, \ldots, N_i^r)$ among all combinations of numbers that add up u. The identifier, called R_i, is an index inside a table F, which contains data on the tuple (see next).
- Information to answer *partial sum* queries on L_i, that is, to compute $\sum_{j=1}^{i} L_j$ in constant time for any block i.
- Information to answer partial sum queries on N_i^c, that is, to compute $\sum_{j=1}^{i} N_j^c$ in constant time for any block i and symbol c.
- For every possible combination $N^1 + \ldots + N^r = u$, a table $E = E_{N^1, \ldots, N^r}$ giving the u-length symbol sequence corresponding to each identifier I_i in that combination. Furthermore, each entry G of E stores the answers to all $\mathsf{Rank}_c(G, q)$ queries, $1 \leq q \leq u$, $c \in [1, r]$.

– A table F, with entries indexed by the R_i numbers, points to the table $E_{N^1,\ldots,N^r}$ corresponding to each valid tuple $(N^1,\ldots,N^r)$.

Solving queries. To answer queries about position q we first compute the block number $i = \lceil q/u \rceil$ where q belongs and the offset $\ell = q-(i-1)u$ inside the block. Then we compute $E = F[R_i]$, the table of entries corresponding to block i, and $G = E[I_i]$, the entry of E corresponding to block i. Note that, since the I_i values use variable number of bits, we need to know which is the starting and ending positions of the representation for I_i in the sequence. These are $1+\sum_{j=1}^{i-1} L_j$ and $\sum_{j=1}^{i} L_j$, respectively, which are known in constant time because we have partial sum information on L_i.

Now, to answer $\mathsf{Rank}_c(S,q)$ we evaluate (in constant time) the partial sum $\sum_{j=1}^{i-1} N_j^c$ and add $\mathsf{Rank}_c(G,\ell)$. To answer $S[q]$ we simply give $G[\ell]$. Both queries take constant time.

Space usage. First notice that $uH_0(S_i) = \log \binom{u}{N_i^1,\ldots,N_i^r}$, and thus

$$\begin{aligned}\sum_{i=1}^{\lceil n/u\rceil} uH_0(S_i) &= \sum_{i=1}^{\lceil n/u\rceil} \log \binom{u}{N_i^1,\ldots,N_i^r} = \log \prod_{i=1}^{\lceil n/u\rceil} \binom{u}{N_i^1,\ldots,N_i^r} \\ &\le \log \binom{n}{n_1,\ldots,n_r} = nH_0(S), \qquad (1)\end{aligned}$$

where n_c is the total number of occurrences of character c in S. The inequality holds because distributing N_i^c symbols over blocks S_i is just one possible way to distribute n_c symbols over S [9]. This result permits bounding the length of the sequence I_i as

$$\begin{aligned}\sum_{i=1}^{\lceil n/u\rceil} L_i &= \sum_{i=1}^{\lceil n/u\rceil} \left\lceil \log \binom{u}{N_i^1,\ldots,N_i^r} \right\rceil \le \sum_{i=1}^{\lceil n/u\rceil} uH_0(S_i) + \lceil n/u \rceil \\ &\le nH_0(S) + O(n/\log_r n).\end{aligned}$$

Let us now consider the numbers R_i. The number of different tuples $(N^1,\ldots,N^r)$ that add up u is $\binom{u+r-1}{r-1} \le (u+1)^r$. Hence it is enough to use $\lceil r\log(u+1)\rceil$ bits for each R_i (which actually is enough to describe any tuple of numbers in $[0,u]$). Accumulated over the $\lceil n/u\rceil$ blocks, this requires $O(rn\log\log n/\log_r n)$ bits for sequence R.

We consider now the structures to answer partial sum queries [9], namely $\sum_{j=1}^{i} N_j^c$ and $\sum_{j=1}^{i} L_j$. Both structures are similar, so we concentrate on L, whose upper bounds are larger: Since $\binom{u}{N_i^1,\ldots,N_i^r} \le r^u$, we

have $0 \leq L_i \leq \lceil u \log r \rceil$. We remark that the length of the sequence is $t = \lceil n/u \rceil$ and the partial sums over L do not exceed $n \lceil \log r \rceil$. Divide L into blocks of length $\lceil \log(n \lceil \log r \rceil) \rceil$, and store the full partial sums for each block beginning. This requires $t = O(n/\log_r n)$ bits. Inside each block, store the partial sums relative to the block beginning. As these cannot exceed $\lceil u \log r \rceil \lceil \log(n \lceil \log r \rceil) \rceil$, we only need $O(\log u + \log\log r + \log\log n) = O(\log\log n)$ bits for each. Hence we need overall $O(t \log\log n) = O(n \log\log n/\log_r n)$ bits for the partial sum information on L. A partial sum query is answered in constant time by adding the partial sum of the block of i and the relative partial sum of i inside its block. The same technique can be applied to sequences N^c, whose values are in the range $[0, u]$, to obtain $O(rn \log\log n/\log_r n)$ bits of space because there are r sequences to index.

Finally, let us consider tables E and F. The total number of entries over all $E_{N^1,\ldots,N^r}$ tables is clearly r^u since each sequence of u symbols over $[1, r]$ belongs exactly to one combination. For each such entry G we store the sequence itself plus answers to all $\mathsf{Rank}_c(G, q)$ queries, needing $O(u \log r + ru \log u)$ bits per entry. Added over all the entries of all the E tables, we have $O(r^u(u \log r + ru \log u)) = O(\sqrt{n} r \log_r n \log\log n)$ bits, which is $o(rn \log\log n/\log_r n)$. Table F has necessarily less entries than E, since there is at least one distinct entry of E for each $(N^1, \ldots, N^r)$ combination in F. Each entry in F points to the corresponding table $E_{N^1,\ldots,N^r}$. If we concatenate all $E_{N^1,\ldots,N^r}$ tables into a supertable of r^u entries, then F points inside that supertable, to the first entry of the corresponding table, and this needs $O(u \log r)$ bits per entry. Overall this adds $O(r^u u \log r)$ bits, which is negligible compared to the size of E.

We remark that the simpler solution of storing pointers $P_i = F[R_i] + I_i$ into the large supertable E would require $n \log r$ bits as the pointers are as long as the sequences represented. This would defeat the whole scheme. Thus we we use table F as an intermediary so as to store the smaller R_i (subtable identifier) and I_i (index into subtable).

Solving $\mathsf{Select}_c(S, q)$ queries. The solution to $\mathsf{Select}_c(S, q)$ queries on binary sequences depicted in [10, Lemma 2.3] divides the sequence into blocks of size u (with the same formula we use for u, with $r = 2$) and makes use of a sequence A, so that A_i is the number of bits set in the i-th block. In our scheme, sequence A corresponds to sequence N^c for each character $c \in [1, r]$. We can use exactly the same scheme of [10] for each of our sequences N^c. They need precisely the same partial sum queries we already considered for N^c, as well as other structures that

require $O(n \log(u)/u)$ bits per sequence N^c. They also need to have all $\mathsf{Select}_c(G, q)$ queries precomputed for each possible block G, which we can add to our E tables for additional $O(r^u ru \log u)$ bits. Overall, the solution needs $O(rn \log(u)/u) = O(rn \log \log n / \log_r n)$ additional bits of space.

Theorem 1. *Let $S[1, n]$ be a sequence of digits in $[1, r]$, with $2 \leq r \leq \sqrt{n}$. There exists a data structure that uses $H_0(S) + O(r(n \log \log n)/ \log_r n)$ bits of space, supporting $S[q]$, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in constant time.*

The theorem is a generalization of the result in [9, 10], which uses $nH_0(S) + O((n \log \log n)/ \log n)$ bits of space to represent a *binary* sequence S ($r = 2$) so as to answer $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in constant time. With constant r, queries $S[q]$ are automatically answered in constant time by finding the c such that $\mathsf{Rank}_c(S, q) - \mathsf{Rank}_c(S, q-1) = 1$. Moreover, with binary sequences we have $R_i = i$.

Note that, although we assume $r \leq \sqrt{n}$, the result is interesting only for $r = o(\log n / \log \log n)$, as otherwise the $O(r(n \log \log n)/ \log_r n)$ term is $\Omega(n \log r)$ and it dominates the space complexity. In the next section we show how to reduce the dependence on r.

3 Generalized wavelet trees

In this section we use the representation of sequences developed in Section 2 to build a more flexible sequence representation, which adapts better to the number of symbols represented. Albeit we solve exactly the same problem, we will change notation a little bit for clarity. This time our sequence $S[1, n]$ will be a sequence of symbols over an *alphabet* $\Sigma = [1, |\Sigma|]$, so that $r \leq \Sigma$ will be reserved to applications of Theorem 1. Actually, for $r = |\Sigma|$ the r-ary wavelet tree will be essentially the structure of Theorem 1.

Structure. Let us consider an *r-ary wavelet tree* built over a string $S[1, n]$. Much like a wavelet tree, each node v of the r-ary tree is responsible for a subset of the alphabet Σ, say Σ_v, so that v represents the subsequence S_v of S formed by the characters in Σ_v. The root node stands for the whole Σ, and thus represents the whole string S. Each node v with children $v_1 \dots v_r$ splits its alphabet Σ_v into r equally-sized subsets $\Sigma_{v_1} \dots \Sigma_{v_r}$, which are integral ranges of size $|\Sigma_{v_i}| \approx |\Sigma_v|/r$. Notice that

the leaves of the tree correspond to singletons of Σ, and that the tree has height at most $1 + \log_r |\Sigma|$.

In each non-leaf node v parent of $v_1, \ldots, v_r$, we represent a sequence S_v, of $n_v = |S_v|$ digits in the range $[1, r]$, such that $S_v[q] = j$ whenever $S_v[q] \in \Sigma_{v_j}$. The data structure of Theorem 1 is built over S_v and stored at node v to answer in constant time $\mathsf{Rank}_j(S_v, q)$ and $S_v[q]$ queries.

To be more precise, all the sequences S_v at each tree level h are concatenated into a long sequence S^h. As we go down the tree, it is easy to maintain in constant time the index $q^* + 1$ where the current node sequence S_v starts inside the level sequence S^h corresponding to node v. Each node v will maintain a vector $C_v[1, r]$, so that $C_v[j]$ is the number of occurrences in S_v of symbols in $[1, j-1]$. If we are currently at node v, whose sequence in level h starts at index $q^* + 1$, and go down to subset Σ_{v_j}, then the sequence for v_j at level $h+1$ starts at $q^* + C_v[j] + 1$. Alternatively, to traverse the tree upwards, if the sequence S_{v_j} starts at index $q^* + 1$ of S^{h+1}, and v is the parent of node v_j, then sequence S_v starts at $q^* - C_v[j] + 1$ in S^h. We can store pointers (with negligible extra space) to find the C vectors of children or parents, or we can take advantage of the tree being almost perfect to avoid such pointers. We need also, for bottom-up traversal, $|\Sigma|$ pointers to the sequences corresponding to each symbol in the bottom sequence S^h.

Solving queries. To compute $\mathsf{Rank}_c(S, q)$, we start at the root node v and determine in constant time the subset Σ_{v_j} to which c belongs. We then compute $q_{v_j} = \mathsf{Rank}_j(S_v, q)$, which is the position corresponding to q in S_{v_j}. We then recursively continue with $q = q_{v_j}$ at node v_j. We eventually reach a tree leaf v_l (corresponding to the subset $\{c\} \subseteq \Sigma$), for which we have the answer to our original query $\mathsf{Rank}_c(S, q) = q_{v_l}$. On the other hand, to determine $S[q]$, we start at the root node v and obtain $j = S_v[q]$, so that $S[q] \in \Sigma_{v_j}$. Then we continue recursively with node v_j and $q = q_{v_j} = \mathsf{Rank}_j(S_v, q)$ as before, until reaching a leaf v_l, where $\Sigma_{v_l} = \{S[q]\}$ is finally determined. Both queries take $O(\log_r |\Sigma|)$ time.

To compute $\mathsf{Select}_c(S, q)$, instead, we proceed bottom-up. We identify the leaf v_l corresponding to subset $\{c\}$ and then proceed upwards. At leaf v_l (not actually represented in the tree), we initialize $q_{v_l} = q$. This is the position we want to track upwards in the tree. Now, let v be the parent of v_j, then $q_v = \mathsf{Select}_j(S_v, q_{v_j})$ is the position of $S_{v_j}[q_{v_j}]$ in S_v. We reach the root, with $q_{root} = \mathsf{Select}_c(S, q)$, in $O(\log_r |\Sigma|)$ time.

Actually we do not represent sequences S_v but level-wise sequences S^h. Assume that our current sequence S_v starts at position $q^* + 1$ in

S^h. Then, queries over S_v are translated to queries over S^h as follows: $S_v[q] = S^h[q^* + q]$, $\mathsf{Rank}_j(S_v, q) = \mathsf{Rank}_j(S^h, q^* + q) - \mathsf{Rank}_j(S^h, q^*)$, and $\mathsf{Select}_j(S_v, q) = \mathsf{Select}_j(S^h, \mathsf{Rank}_j(S^h, q^*) + q)$.

Space usage. An immediate advantage of having all sequences $S^h[1, n]$ over the same alphabet $[1, r]$ is that all tables E and F are the same for all levels, so they take $o((rn \log\log n)/\log_r n)$ overall. All the other $O((rn \log\log n)/\log_r n)$ size structures used to prove Theorem 1 totalize $O(\log|\Sigma|\,(rn \log\log n)/\log n)$ bits of space by adding up all the levels. The new structures C_v we introduced need $O(r \log n)$ bits each, and there is one C_v array per non-leaf node v. This totalizes $O\left(\frac{|\Sigma|}{r-1} r \log n\right) = O(|\Sigma| \log n)$ bits. This space includes also the pointers to leaves and the parent-child pointers in the tree, if they are used.

Let us consider now the entropy-related part. For each non-leaf node v at tree level h, with children $v_1, \ldots, v_r$, sequence S_v spans at most $2 + \lfloor n_v/u \rfloor$ blocks in S^h (recall from Section 2 that the sequence is divided into blocks of length $u = \left\lfloor \frac{1}{2} \log n \right\rfloor$). The sum of local zero-order entropies $uH_0(S_i^h)$ for the $\lfloor n_v/u \rfloor$ blocks is a lower bound to $n_v H_0(S_v)$ (recall Eq. (1)). For the other 2 blocks, we simply assume that they take the maximum $u \lceil \log r \rceil$ bits. Added over all the sequence boundaries over the whole tree we have at most $\frac{r}{r-1}|\Sigma|$ boundaries, for an overall space overhead of $O(|\Sigma| \log n)$ bits due to partial blocks.

Thus, let us focus on the summation of all $n_v H_0(S_v)$ terms over all the tree. Note that this is

$$\begin{aligned} -n_v \sum_{j=1}^{r} \frac{n_{v_j}}{n_v} \log \frac{n_{v_j}}{n_v} &= -\sum_{j=1}^{r} n_{v_j} \log n_{v_j} + \sum_{j=1}^{r} n_{v_j} \log n_v \\ &= n_v \log n_v - \sum_{j=1}^{r} n_{v_j} \log n_{v_j}, \end{aligned}$$

where we note that the addition of all the first terms $n_{v_j} \log n_{v_j}$ over all the children of v will cancel with the second term of the formula for their parent v. Therefore, after all the cancelations, the only surviving terms are $n \log n$ corresponding to the tree root and the terms $-n_{u_l} \log n_{u_l}$ corresponding to the parents of the tree leaves (where $n_{u_l} = n_c$ for some $c \in \Sigma$, being n_c the frequency of character c). This is

$$n \log n - \sum_{c \in \Sigma} n_c \log(n_c) = nH_0(S).$$

Theorem 2. *Let $S[1,n]$ denote a string over an arbitrary alphabet Σ. The r-ary wavelet tree built on S, for $2 \leq r \leq \min(|\Sigma|, \sqrt{n})$, uses $nH_0(S) + O(|\Sigma| \log n) + O(\log |\Sigma|\, r(n \log\log n)/\log n)$ bits of storage and supports in $O(\log_r |\Sigma|)$ time the queries $S[q]$, $\mathsf{Rank}_c(S,q)$ and $\mathsf{Select}_c(S,q)$, for any $c \in \Sigma$ and $1 \leq q \leq n$.*

Moreover, if $|\Sigma| = O(\mathit{polylog}(n))$, then r can be chosen so that the tree answers both queries in constant time and takes $nH_0(S) + O(n/\log^{1-\varepsilon} n)$ bits of space, for any constant $\varepsilon > 0$.

Proof. The first part of the theorem, for general r, is a consequence of the development in this section. For the last sentence, note that by choosing $r = |\Sigma|^{1/\kappa}$, for constant $\kappa > 0$, we can support the query operations in constant time $O(\kappa)$. Now, if $|\Sigma| = O(\mathrm{polylog}(n)) = O((\log n)^d)$, then we can choose any $\kappa > d$ to obtain $O(\kappa n (\log\log n)^2/(\log n)^{1-d/\kappa})$ space overhead. For any constant $\varepsilon > 0$, we choose $k > d/\varepsilon$ to ensure that $O(n(\log\log n)^2/(\log n)^{1-d/\kappa}) = O(n/\log^{1-\varepsilon} n)$. □

The theorem is a generalization upon the (binary) wavelet tree data structure [4], which takes $nH_0(S) + O(\log |\Sigma|\ (n \log\log n)/\log n)$ space and answers the same queries in $O(\log |\Sigma|)$ time. The last part shows that the generalization permits an improvement when $|\Sigma| = O(\mathrm{polylog}(n))$.

4 Full-text indexes

Given a string $T[1,n]$ of symbols over an alphabet Σ, a full-text index over T supports the following queries: (i) count the number of occurrences of a pattern string $P[1,p]$ as a substring of T, (ii) locate the position in T of each such occurrence, (iii) retrieve $T[i, i+\ell-1]$, a substring of T of length ℓ. The following theorem was proved in [2]:

Theorem 3. *Let $T[1,n]$ be a string over an alphabet Σ. Then there exists a representation for T using $nH_k(T) + O(\log |\Sigma| (n \log\log n)/\log n) + O(n/\log^{\varepsilon} n)$ bits for any $k \leq \alpha \log_{|\Sigma|} n$, $0 < \alpha < 1$, and $\varepsilon > 0$, such that it can count the number of occurrences of any pattern string $P[1,p]$ in T in $O(p \log |\Sigma|)$ time, locate each occurrence in $O(\log |\Sigma| \log^{1+\varepsilon} n)$ time, and retrieve any text substring of length ℓ in $O((\ell + \log^{1+\varepsilon} n) \log |\Sigma|)$ time.*

The result was obtained using wavelet trees [4]. Actually, it was shown in [2] that any alternative to the wavelet tree can be used to obtain different space-time tradeoffs, as follows.

Theorem 4. *Let $T[1,n]$ be a string over an alphabet Σ. Assume there exists a representation of sequences $S[1,s]$ of symbols in $[1,|\Sigma|]$ taking $sH_0(S)+f(s)$ bits of space, for concave f, which answers queries $S[q]$ in time $t_a(s)$ and $\mathsf{Rank}_c(S,q)$ in time $t_r(s)$. Then there exists a representation for T using $nH_k(T)+f(n)+O(n/\log^\varepsilon n)$ bits, for any $k \leq \alpha \log_{|\Sigma|} n$, $0<\alpha<1$, and $\varepsilon>0$, such that it can count the number of occurrences of a pattern string $P[1,p]$ in T in $O(p\ t_r(n))$ time, locate each occurrence in $O((t_a(n)+t_r(n))\log^{1+\varepsilon} n)$ time, and retrieve any text substring of length ℓ in $O((\ell+\log^{1+\varepsilon} n)(t_a(n)+t_r(n)))$ time.*

By combining Theorems 2 and 4, and using an upper limit of 1/2 for both ε's, we obtain the following:

Theorem 5. *Let $T[1,n]$ be a string over an alphabet Σ, where $|\Sigma| = O(polylog(n))$. Then there exists a representation for T using $nH_k(T) + O(n/\log^\varepsilon n)$ bits for any $k \leq \alpha \log_{|\Sigma|} n$, $0<\alpha<1$, and $0<\varepsilon<1/2$, such that it can count the number of occurrences of any pattern string $P[1,p]$ in T in $O(p)$ time, locate each occurrence in $O(\log^{1+\varepsilon} n)$ time, and retrieve any text substring of length ℓ in $O(\ell+\log^{1+\varepsilon} n)$ time.*

This is the best existing result on succinct full-text indexes, in the sense that our index belongs to the class of the most succinct existing indexes [2, 4], and it is the fastest among those.

5 Conclusions

We have presented a new succinct sequence representation that takes space essentially equal to its zero-order entropy. This representation permits retrieving any sequence character, knowing how many times any character has appeared before any position, and finding the q-th occurrence of any character, all in constant time. Our result extends previous results on binary sequences [9, 10] and improves previous results that answers those queries in time logarithmic with the maximum number appearing in the sequence [4].

We have shown that our representation can be applied to improve a recent result on full-text indexing [2]. The text index we present belongs to the class of the most succinct ones, needing essentially the space for a k-th order entropy representation of the text. Among those indexes, ours gives the best search complexities.

References

1. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
2. 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 2004)*, LNCS, 2004. To appear.
3. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Combinatorial Pattern Matching Conference (CPM'04)*, LNCS 3109, pages 159–172, 2004.
4. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.
5. G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.
6. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer-Verlag LNCS n. 1180, 1996.
7. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.
8. J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, pages 345–356. Springer-Verlag LNCS n. 2719, 2003.
9. R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. ICALP*, pages 595–604, 1999.
10. R. Raman, V. Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 233–242, 2002.