Theoretical Computer Science 513 (2013) 109–123

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On compressing permutations and adaptive sorting ☆

Jérémy Barbay [1], Gonzalo Navarro *,[2]

Dept. of Computer Science, University of Chile, Chile

ARTICLE INFO

Article history:
Received 6 May 2013
Received in revised form 29 September 2013
Accepted 7 October 2013
Communicated by G.F. Italiano

Keywords:
Compression
Permutations
Succinct data structures
Adaptive sorting

ABSTRACT

We prove that, given a permutation π over $[1..n]$ formed of $\mathtt{nRuns}$ sorted blocks of sizes given by the vector $R = \langle r_1, \ldots, r_{\mathtt{nRuns}} \rangle$, there exists a compressed data structure encoding π in $n(1+\mathcal{H}(R)) = n + \sum_{i=1}^{\mathtt{nRuns}} r_i \log_2 \frac{n}{r_i} \leqslant n(1+\log_2 \mathtt{nRuns})$ bits while supporting access to the values of $\pi()$ and $\pi^{-1}()$ in time $\mathcal{O}(\log \mathtt{nRuns}/\log\log n)$ in the worst case and $\mathcal{O}(\mathcal{H}(R)/\log\log n)$ on average, when the argument is uniformly distributed over $[1..n]$. This data structure can be constructed in time $\mathcal{O}(n(1+\mathcal{H}(R)))$, which yields an improved adaptive sorting algorithm. Similar results on compressed data structures for permutations and adaptive sorting algorithms are proved for other preorder measures of practical and theoretical interest.

1. Introduction

Permutations of the integers $[1..n] = \{1, \ldots, n\}$ are a fundamental mathematical structure, and a basic building block for the succinct encoding of integer functions [39], strings [30,22,25,2,34,14], binary relations [9], and geometric grids [13], among others. A permutation π can be trivially encoded in $n\lceil \lg n \rceil$ bits, which is within $\mathcal{O}(n)$ bits of the information theory lower bound of $\lg(n!)$ bits, where $\lg x = \log_2 x$ denotes the logarithm in base two.

Efficient computation for both the value $\pi(i)$ at any point $i \in [1..n]$ of the permutation, and for the position $\pi^{-1}(j)$ of any value $j \in [1..n]$ (i.e., the value of the inverse permutation) is essential in most of those applications. A trivial solution is to store explicitly both π and π^{-1}, using a total of $2n\lceil \lg n \rceil$ bits. Munro et al. [39] proposed three nontrivial alternatives. The first consists in plainly representing π in $n\lceil \lg n \rceil$ bits (hence supporting the operator $\pi()$ in constant time) and adding a small structure of $\epsilon\, n \lg n$ extra bits in order to support the operator $\pi^{-1}()$ in time $\mathcal{O}(1/\epsilon)$. The second solution uses the previous one to encode another permutation, the one mapping the original permutation to a cycle representation, which yields support for any positive or negative power of $\pi()$, $\pi^k(i)$ for any $k \in \mathbb{Z}$. The third solution uses less space (only $\mathcal{O}(n)$ extra bits, as opposed to $\epsilon\, n \lg n$) but supports the operator $\pi^k(j)$ for any value of k and j in higher time, within $\mathcal{O}(\log n/\log\log n)$. Each of those solutions uses at least $\lceil n \log_2 n \rceil$ bits to encode the permutation itself.

The lower bound of $\lg(n!)$ bits to represent any permutation yields a lower bound of $\Omega(n \log n)$ comparisons to sort a permutation in the comparison model, in the worst case over all permutations of n elements. A large body of research has been dedicated to finding better sorting algorithms that can take advantage of specificities of certain families of permutations. Some examples are permutations composed of a few sorted blocks (also called "runs") [35] (e.g., $(1, 3, 5, 7, 9, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ or $(6, 7, 8, 9, 10, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$), or permutations containing few sorted subsequences [33] (e.g., $(1, \mathbf{6}, 2, \mathbf{7}, 3, \mathbf{8}, 4, \mathbf{9}, 5, \mathbf{10})$). Algorithms performing possibly $o(n \log n)$ comparisons on such permutations, yet still $\mathcal{O}(n \log n)$ comparisons in the worst

☆ An early version of this article appeared in *Proc. STACS 2009* [10].

* Corresponding author.
E-mail addresses: jbarbay@dcc.uchile.cl (J. Barbay), gnavarro@dcc.uchile.cl (G. Navarro).

[1] Partially funded by Fondecyt Grant 1120054, Chile.
[2] Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.10.019

case, are achievable and preferable if those permutations arise with sufficient frequency. Other examples are classes of permutations whose structure makes them interesting for applications; see the seminal paper of Mannila [35], and the survey of Moffat and Petersson [37].

Each sorting algorithm in the comparison model yields an encoding scheme for permutations: the result of all the comparisons performed uniquely identifies the permutation sorted, and hence encodes it. Since an adaptive sorting algorithm performs $o(n \log n)$ comparisons on a class of "easy" permutations, each adaptive algorithm yields a *compression scheme* for permutations, at the cost of losing a constant factor on the complementary class of "hard" permutations. Yet such compression schemes do not necessarily support efficiently the computation of arbitrary $\pi(i)$ values, nor the inverse permutation values $\pi^{-1}(j)$.

It is natural to ask whether it is possible to compress a permutation π [37] while at the same time supporting efficient access to π and its inverse [39]. To the best of our knowledge, such a representation had not been described till now. In this paper we describe a whole family of such compressed data structures, inspired by and improving upon the `MergeSort` family of adaptive sorting algorithms [35]. All of them take advantage of permutations composed of a small number of monotone subsequences, and support the operators $\pi()$ and $\pi^{-1}()$ efficiently, taking less time on the more compressible permutations.

Our central result (Theorem 3) is a compressed data structure based on the decomposition of a permutation π into "runs", that is, monotone subsequences of consecutive positions. If π is formed by $\mathtt{nRuns}$ runs of sizes given by the vector $R = \langle r_1, \ldots, r_{\mathtt{nRuns}} \rangle$, our data structure encodes it in $n(1 + \mathcal{H}(R)) = n + \sum_{i=1}^{\mathtt{nRuns}} r_i \lg \frac{n}{r_i} \leqslant n(1 + \lg \mathtt{nRuns})$ bits and supports access to the values of $\pi()$ and $\pi^{-1}()$ in time $\mathcal{O}(\log \mathtt{nRuns} / \log\log n)$ in the worst case and $\mathcal{O}(\mathcal{H}(R)/\log\log n)$ on average, when the argument is uniformly distributed over $[1..n]$. The construction of this data structure yields an improved adaptive sorting algorithm running in time $\mathcal{O}(n(1 + \mathcal{H}(R)))$. Similar data structures and adaptive sorting algorithms are obtained, via reductions, for other preorder measures of practical and theoretical interest, such as "strict runs", a particular case of runs with consecutive values, and "shuffled sequences", monotone subsequences of not necessarily consecutive positions. Those results have applications to the indexing of natural language text collections, the support of compressed suffix arrays, and the representation of strings supporting operations access, rank, and select (Theorem 8). The latter result improves upon the state of the art [16,23] in the average case when the queries are uniformly distributed, while retaining the space and worst-case performance of the previous solutions.

2. Basic concepts and previous work

For completeness, we review here some basic notions and techniques about entropy (Section 2.1), Huffman codes (Section 2.2), data structures on sequences (Section 2.3) and adaptive sorting algorithms (Section 2.4). Readers already familiar with those notions can safely skip this section.

2.1. Entropy

We define the *entropy* of a distribution [15], a measure that will be useful to evaluate compressibility results.

Definition 1. The *entropy* of a sequence of positive integers $X = \langle n_1, n_2, \ldots, n_r \rangle$ adding up to n is $\mathcal{H}(X) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$. By concavity of the logarithm, it holds that $(r-1)\lg n \leqslant n\mathcal{H}(X) \leqslant n \lg r$ and that $\mathcal{H}(\langle n_1, n_2, \ldots n_r \rangle) > \mathcal{H}(\langle n_1 + n_2, \ldots, n_r \rangle)$.

Here $X = \langle n_1, n_2, \ldots, n_r \rangle$ is a distribution and $\mathcal{H}(X)$ measures how even is it. $\mathcal{H}(X)$ is maximal ($\lg r$) when all $n_i = n/r$ and minimal ($\frac{r-1}{n} \lg n + \frac{n-r+1}{n} \lg \frac{n}{n-r+1}$) when they are most skewed ($X = \langle 1, 1, \ldots, 1, n-r+1 \rangle$).

This measure is related to the entropy of random variables and of sequences as follows. If a random variable P takes the value i with probability n_i/n, for $1 \leqslant i \leqslant r$, then its entropy is $\mathcal{H}(\langle n_1, n_2, \ldots, n_r \rangle)$. Similarly, if a string $S[1..n]$ contains n_i occurrences of character c_i, then its empirical zero-order entropy is $\mathcal{H}_0(S) = \mathcal{H}(\langle n_1, n_2, \ldots, n_r \rangle)$.

$\mathcal{H}(X)$ is then a lower bound to the average number of bits needed to encode an instance of P, or to encode a character of S (if we model S statistically with a zero-order model, that is, ignoring the context of characters).

2.2. Huffman codes

Given symbols $[1..r]$ with frequencies $X = \langle n_1, n_2, \ldots, n_r \rangle$ adding up to n, Huffman [28] described how to build an optimal prefix-free code for them. His algorithm can be implemented in time $\mathcal{O}(r \log r)$. If ℓ_i is the bit length of the code assigned to the ith symbol, then $L = \sum \ell_i n_i$ is minimal and $L < n(1 + \mathcal{H}(X))$. For example, given a string $S[1..n]$ over alphabet $[1..r]$, with symbol frequencies $X[1..r]$, one can compress S by concatenating the codewords of the successive symbols $S[i]$, achieving total length $L < n(1 + \mathcal{H}_0(S))$. (One also has to encode the usually negligible codebook of $\mathcal{O}(r \log r)$ bits.)

The algorithm to build the optimal prefix-free code starts with a forest of r leaves corresponding to the frequencies $\{n_1, n_2, \ldots, n_r\}$, and outputs a binary trie with those leaves, in some order. This so-called *Huffman tree* describes the optimal encoding as follows: The sequence of left/right choices (interpreted as 0/1) in the path from the root to each leaf is the prefix-free encoding of that leaf, of length ℓ_i equal to the leaf depth.

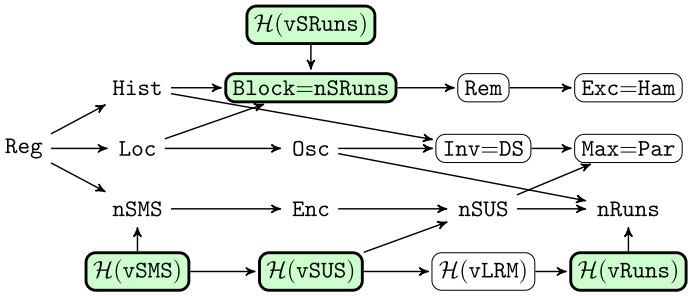


Fig. 1. Partial order on some measures of disorder for adaptive sorting, completed from Moffat and Petersson's 1992 survey [37]. Round boxes signal the measures for which new results have been proved since then (all inspired by our results), and bold ones signal the results introduced in this article. A measure A dominates a measure B ($A \to B$) if all optimal algorithms for A have a better asymptotic complexity (i.e., for instances large enough and ignoring constant factors) than some optimal algorithms for B. In this sense, the measures $\mathcal{H}(\mathtt{vSMS})$ and $\mathcal{H}(\mathtt{vSUS})$ are of theoretical interest because their asymptotic complexities involve larger constant factors, while the measures $\mathcal{H}(\mathtt{vRuns})$ and $\mathcal{H}(\mathtt{vSRuns})$ are more practical. The measure $\mathtt{nSRuns}$ is presented for completeness and the measure $\mathcal{H}(\mathtt{vLRM})$, not presented in this work, is a side result of another technique [7] (see Section 6.4).

A generalization of this encoding is *multiary Huffman coding* [28], in which the tree is given arity t, and then the Huffman codewords are sequences over an alphabet $[1..t]$. In this case the algorithm also produces the optimal t-ary code, of length $L < n(1 + \mathcal{H}(X)/\lg t)$.

2.3. Succinct data structures for sequences

Let $S[1..n]$ be a sequence of symbols from the alphabet $[1..r]$. This includes bitmaps when $r = 2$ (where, for convenience, the alphabet will be $\{0, 1\}$ rather than $\{1, 2\}$). We will make use of succinct representations of S that support the rank and select operators over strings and over binary vectors: $\mathtt{rank}_c(S, i)$ gives the number of occurrences of c in $S[1..i]$ and $\mathtt{select}_c(S, j)$ gives the position in S of the jth occurrence of c.

When $r = 2$, S requires n bits and $\mathtt{rank}$ and $\mathtt{select}$ can be supported in constant time using $\mathcal{O}(n \log\log n/\log n) \subset o(n)$ bits on top of S [38,21].

Raman et al. [43] devised a bitmap representation that takes $n\mathcal{H}_0(S) + o(n)$ bits, while maintaining the constant time for supporting the operators. For the binary case $\mathcal{H}_0(S)$ is just $m \lg \frac{n}{m} + (n-m)\lg \frac{n}{n-m} \in m \lg \frac{n}{m} + \mathcal{O}(m)$, where m is the number of bits set to 1 in S. Pătraşcu [42] reduced the $o(n)$-bits redundancy in space to $\mathcal{O}(n/\log^c n)$ for any constant c (we will use just $c = 2$ in this paper).

When m is much smaller than n, the $o(n)$-bits term may dominate. Gupta et al. [27] showed how to achieve space within $m \lg \frac{n}{m} + \mathcal{O}(m \log\log \frac{n}{m} + \log n)$ bits, which largely reduces the dependence on n, but now $\mathtt{rank}$ and $\mathtt{select}$ are supported in time $\mathcal{O}(\log m)$ via binary search [26, Theorem 17, p. 153].

For larger alphabets, of size $r \in o(\log n)$, Ferragina et al. [16] showed how to represent the sequence within $n\mathcal{H}_0(S) + o(n \log r)$ bits and support $\mathtt{rank}$ and $\mathtt{select}$ in constant time. Golynski et al. [23, Lemma 9] improved the space to $n\mathcal{H}_0(S) + o(n \log r/\log n)$ bits while retaining constant times.

Grossi et al. [24] introduced the *wavelet tree*, which decomposes a sequence over an alphabet of arbitrary size r into several bitmaps. By representing the bitmaps in compressed form [42], the overall space is $n\mathcal{H}_0(S) + o(n)$ and $\mathtt{rank}$ and $\mathtt{select}$ are supported in time $\mathcal{O}(\log r)$. Multiary wavelet trees decompose the sequence into subsequences over a sublogarithmic-sized alphabet and reduce the time to $\mathcal{O}(1 + \log r/\log\log n)$ while retaining space $n\mathcal{H}_0(S) + o(n)$ [16,23].

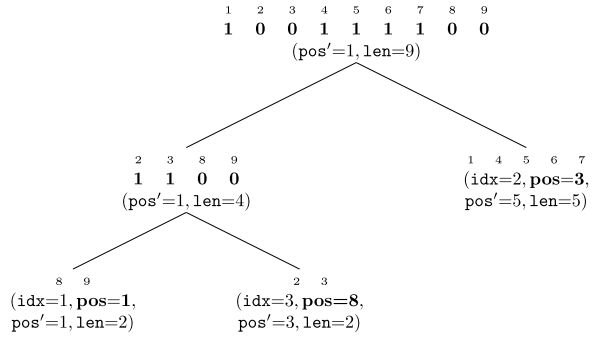
In this article n will generally denote the length of the permutation. All of our $o()$ expressions, even those with several variables, will be asymptotic in n.

2.4. Measures of presortedness in permutations

The complexity of *adaptive algorithms*, for problems such as searching, sorting, merging sorted arrays or convex hulls, is studied in the worst case over instances of fixed size *and difficulty*, for a definition of difficulty that is specific to each analysis. Even though sorting a permutation in the comparison model requires $\Theta(n \log n)$ comparisons in the worst case over all the permutations of n elements, better results can be achieved for some parameterized classes of permutations. We describe some of those below, see the survey of Moffat and Petersson [37] for other results.

Knuth [32] considered *runs* (contiguous ascending subsequences) of a permutation π, counted by $\mathtt{nRuns} = 1 + |\{i\colon\ 1 \leqslant i < n,\ \pi(i+1) < \pi(i)\}|$. Levcopoulos and Petersson [33] introduced *Shuffled UpSequences* and its generalization *Shuffled Monotone Sequences*, respectively counted by $\mathtt{nSUS} = \min\{k : \pi$ is covered by k increasing subsequences$\}$, and $\mathtt{nSMS} = \min\{k\colon\ \pi$ is covered by k monotone subsequences$\}$. By definition, $\mathtt{nSMS} \leqslant \mathtt{nSUS} \leqslant \mathtt{nRuns}$. The relations between those preorder measures, others not described here, and new ones described in this article, are represented in Fig. 1.

Munro and Spira [40] took an orthogonal approach, considering the problem of sorting multisets through various algorithms such as $\mathtt{MergeSort}$. They showed that the algorithms can be adapted to run in time $\mathcal{O}(n(1 + \mathcal{H}(\langle m_1, \ldots, m_r\rangle)))$ where m_i is the number of occurrences of i in the multiset (note this is totally different from our results, which depend on the distribution of the lengths of monotone runs).



$$\begin{aligned}\pi &= (8,9,1,4,5,6,7,2,3)\\ \texttt{nRuns} &= 3\\ \texttt{vRuns} &= \langle 2,5,2\rangle\\ \mathbf{C} &= \mathbf{101000010}\\ \phi &= \langle \mathbf{1},\mathbf{3},\mathbf{2}\rangle\\ \pi' &= (8,9,2,3,1,4,5,6,7)\end{aligned}$$

Fig. 2. Example of the runs-compressed data structure, highlighting in bold which of the variables computed during the compression represent the permutation in the end.

Each adaptive sorting algorithm in the comparison model yields a compression scheme for permutations, but the encoding thus defined does not necessarily support the simple application of the permutation to a single element without decompressing the whole permutation, nor the application of its inverse.

3. Contiguous monotone runs

Our most fundamental representation takes advantage of permutations that are formed by a few monotone (ascending or descending) runs.

Definition 2. A *down-step* of a permutation π over $[1..n]$ is a position $1 \leqslant i < n$ such that $\pi(i+1) < \pi(i)$. An *ascending run* in a permutation π is a maximal range of consecutive positions $[i..j]$ that does not contain any down-step. Let $d_1, d_2, \ldots, d_k$ be the list of consecutive down-steps in π. Then the number of ascending runs of π is denoted by $\texttt{nRuns} = k+1$, and the sequence of the lengths of the ascending runs is denoted by $\texttt{vRuns} = \langle n_1, n_2, \ldots, n_{\texttt{nRuns}}\rangle$, where $n_1 = d_1, n_2 = d_2 - d_1$, $\ldots, n_{\texttt{nRuns}-1} = d_k - d_{k-1}$, and $n_{\texttt{nRuns}} = n - d_k$. (If $k = 0$ then $\texttt{nRuns} = 1$ and $\texttt{vRuns} = \langle n_1\rangle = \langle n\rangle$.) The notions of *up step* and *descending run* are defined similarly.

For example, the permutation $(8, 9, 1, 4, 5, 6, 7, \mathbf{2}, \mathbf{3})$ of Fig. 2 contains $\texttt{nRuns} = 3$ ascending runs, of lengths forming the vector $\texttt{vRuns} = \langle 2, 5, 2\rangle$. We now describe a data structure that represents a permutation partitioned into $\texttt{nRuns}$ ascending runs, and is able to support any $\pi(i)$ and $\pi^{-1}(i)$ efficiently.

3.1. Structure

Consider the sorting algorithm `MergeSort`. Its merging process can be represented as a balanced binary tree of height $\lg n$. Detecting runs and merging them pairwise and hierarchically makes `MergeSort` adaptive to the number `nRuns` of runs. The reduced merging process is then represented by a balanced binary tree of height $\lg \texttt{nRuns}$ and the total sorting time becomes $\mathcal{O}(n + n \log \texttt{nRuns})$. Merging the two shortest runs at each step further improves `MergeSort`, making its running time adaptive to the entropy of the vector `vRuns` formed by the lengths of the runs, $\mathcal{O}(n + \mathcal{H}(\texttt{vRuns}))$. The merging process is then represented by a tree with the same shape of a Huffman tree for the distribution `vRuns`. Keeping the result of each comparison performed by those algorithms yields a compressed encoding of the permutation that identifies it uniquely. To support forward and inverse access to the individual values of π in less time than required to uncompress the whole encoding, it is enough to memorize the lengths of the runs and their reordering into the leaves of the merging tree.

3.1.1. Construction

We find the down-steps of π in linear time, obtaining $\texttt{nRuns}$ runs of lengths $\texttt{vRuns} = \langle n_1, \ldots, n_{\texttt{nRuns}}\rangle$, and then apply the Huffman algorithm to the vector $\texttt{vRuns}$. When we set up the leaves v of the Huffman tree, we store their original index in $\texttt{vRuns}$, $\texttt{idx}(v)$, the starting position in π of their corresponding run, $\texttt{pos}(v)$, and the length of their run, $\texttt{len}(v)$. After the tree is built, we use $\texttt{idx}(v)$ to compute a permutation ϕ over $[1..\texttt{nRuns}]$ so that $\phi(i) = j$ if the leaf corresponding to n_i is placed at the jth left-to-right leaf in the Huffman tree. We also precompute a bitmap $C[1..n]$ that marks the beginning of runs in π, with constant-time support for `rank` and `select`. Since C contains only $\texttt{nRuns}$ bits set out of n, it is represented in compressed form [43] within $\texttt{nRuns} \lg \frac{n}{\texttt{nRuns}} + \mathcal{O}(\texttt{nRuns}) + o(n)$ bits.

Now we set a new permutation π' over $[1..n]$ where the runs are written in the order given by ϕ^{-1}: We first copy from π the run whose endpoints are those of the leftmost tree leaf, then the run pointed by the second leftmost leaf, and so on. The endpoints of the runs are obtained with $\texttt{pos}(v)$ and $\texttt{len}(v)$. Simultaneously, we create field $\texttt{pos}'(v)$ as the starting position of the area v covers in π'. After creating π' the original permutation π can be deleted. We say that an internal

The value $\pi^{-1}(9)$

- is computed by navigating T top-down:
- the 9-th bit in the top bitmap is the 4th 0,
- the 4-th bit of the left child is the 2nd 0.
- We reach offset 2 in the first leaf v of T.
- Hence $\pi^{-1}(9) = \texttt{pos}(v) + 2 - 1 = 2$.

(As can be checked in Figure 2.)

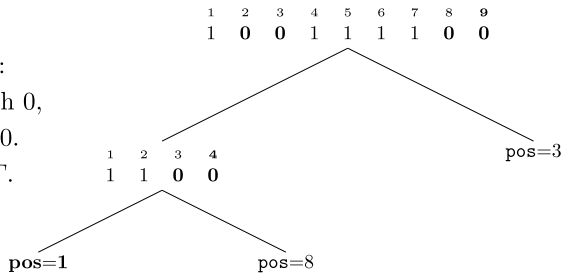


Fig. 3. Computing $\pi^{-1}()$ on the runs-compressed data structure, using the example permutation of Fig. 2. We mark in bold the bits counted in the `rank` operations.

node *covers* the contiguous area of π' formed by concatenating the runs of all the leaves that descend from v. We propagate the leaf `pos`$'$ and `len` values to all the internal nodes v, so that `pos`$'(v)$ is the starting position of the area covered by v in π', and `len`(v) is the length of that area.

Now we enhance the Huffman tree into a wavelet-tree-like structure [24] without altering its shape, as follows. Starting from the root, first process recursively each child. For the leaves we do nothing. Once the left and right children, v_l and v_r, of an internal node v have been processed, the invariant is that the areas they cover have already been sorted in π'. We create a bitmap for v, of size `len`(v). Now we merge the areas of v_l and v_r in time $\mathcal{O}(\texttt{len}(v))$. As we do the merging, each time we take an element from v_l we append a bit 0 to the node bitmap, and a bit 1 when we take an element from v_r. When we finish, π' has been sorted and we can delete it. The Huffman-shaped wavelet tree (only with the bitmaps and field `pos`, but storing `nRuns` pointers to the leaves and parent pointers), ϕ, and C, represent π. See Fig. 2 for an example.

3.1.2. Space and construction cost

Note that each of the n_i elements of leaf i (at depth ℓ_i) is merged ℓ_i times, contributing ℓ_i bits to the bitmaps of its ancestors, and thus the total number of bits in all bitmaps is $\sum n_i \ell_i$. Therefore, the total number of bits in the Huffman-shaped wavelet tree is at most $n(1 + \mathcal{H}(\texttt{vRuns}))$. Those bitmaps, however, are represented in compressed form [42], which allows us to remove the n extra bits added by the Huffman encoding.

Let us call $m_j = n_{\phi^{-1}(j)}$ the length of the run corresponding to the jth left-to-right leaf, and $m_{i,j} = m_i + \cdots + m_j$. The compressed representation [42] takes, on a bitmap of length n and m 1s, $m \lg \frac{n}{m} + (n-m) \lg \frac{n}{n-m}$ bits, plus a redundancy of $\mathcal{O}(n/\log^2 n)$ bits. We prove by induction (see also Grossi et al. [24]) that the compressed space allocated for all the bitmaps descending from a node covering leaves $[i..k]$ is $\sum_{i \leqslant r \leqslant k} m_r \lg \frac{m_{i,k}}{m_r}$ (we consider the redundancy later). Consider two sibling leaves merging two runs of m_i and m_{i+1} elements. Their parent bitmap contains m_i 0s and m_{i+1} 1s, and thus its compressed representation requires $m_i \lg \frac{m_i + m_{i+1}}{m_i} + m_{i+1} \lg \frac{m_i + m_{i+1}}{m_{i+1}}$ bits. Now consider a general Huffman tree node merging a left subtree covering leaves $[i..j]$ and a right subtree covering leaves $[j+1..k]$. Then the bitmap of the node will be compressed to $m_{i,j} \lg \frac{m_{i,k}}{m_{i,j}} + m_{j+1,k} \lg \frac{m_{i,k}}{m_{j+1,k}}$ bits. By the inductive hypothesis, all the bitmaps on the left child and its subtrees add up to $\sum_{i \leqslant r \leqslant j} m_r \lg \frac{m_{i,j}}{m_r}$, and those on the right add up to $\sum_{j+1 \leqslant r \leqslant k} m_r \lg \frac{m_{j+1,k}}{m_r}$. Adding up the three formulas we get the inductive thesis.

Therefore, a compressed representation of the bitmaps requires $n\mathcal{H}(\texttt{vRuns})$ bits, plus the redundancy. The latter, added over all the bitmaps, is $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns}))/\log^2 n) \subset o(n)$ because $\mathcal{H}(\texttt{vRuns}) \leqslant \lg n$. To this we must add the $\mathcal{O}(\texttt{nRuns} \log n)$ bits of the tree pointers, bitmap pointers and lengths, fields `pos`, the permutation ϕ, and the bitmap C.

The construction time is $\mathcal{O}(\texttt{nRuns} \log \texttt{nRuns})$ for the Huffman algorithm, plus $\mathcal{O}(\texttt{nRuns})$ for computing ϕ and filling the node fields `idx`, `pos`, `len` and `pos`$'$, plus $\mathcal{O}(n)$ for constructing π' and C, plus the total number of bits appended to all the bitmaps, which includes the merging cost. The extra structures for `rank` are built in linear time on those bitmaps. All this adds up to $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})))$, because $\texttt{nRuns} \lg \texttt{nRuns} \leqslant n\mathcal{H}(\texttt{vRuns}) + \lg n$ by concavity, recall Definition 1.

3.2. Queries

3.2.1. Computing $\pi()$ and $\pi^{-1}()$

One can regard the wavelet tree as a device that tracks the evolution of a merge-sorting of π', so that in the bottom we have (conceptually) the sequence π' (with one run per leaf) and in the top we have (conceptually) the sorted permutation $(1, 2, \ldots, n)$.

To compute $\pi^{-1}(j)$ for any $j \in [1..n]$ we start at the top and find out where that position came from in π'. We start at offset $j' = j$ of the root bitmap B. If $B[j'] = 0$, then position j' came from the left subtree in the merging. Thus we go down to the left child with $j' \leftarrow \texttt{rank}_0(B, j')$, which is the position of j' in the array of the left child before the merging. Otherwise we go down to the right child with $j' \leftarrow \texttt{rank}_1(B, j')$. We continue recursively until we reach a leaf v. At this point we know that j came from the corresponding run, at offset j', that is, $\pi^{-1}(j) = \texttt{pos}(v) + j' - 1$. See Fig. 3 for an example.

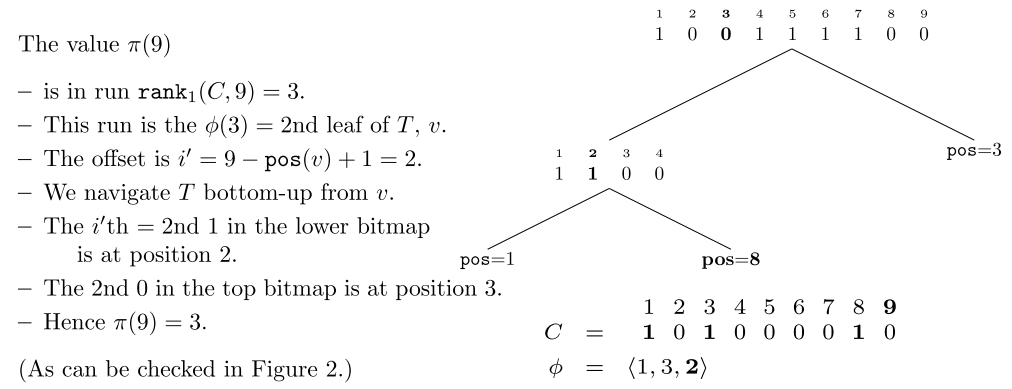


Fig. 4. Example of support of $\pi()$ on a Runs-compressed Data Structure, using the same permutation as in Fig. 2. We mark in bold the bits counted in the `rank` operations.

To compute $\pi(i)$ for any $i \in [1..n]$ we do the reverse process, but we must first determine the leaf v and offset i' within v corresponding to position i. We compute $l = \phi(\texttt{rank}_1(C, i))$, so that i falls at the lth left-to-right leaf. Then v is the lth entry in our array of pointers to the leaves, and the offset is $i' = i - \texttt{pos}(v) + 1$. We now start an upward traversal from v using the parent pointers. If v is the left child of its parent u, then we set $i' \leftarrow \texttt{select}_0(B, i')$ to locate it in the merged array of the parent, else we set $i' \leftarrow \texttt{select}_1(B, i')$, where B is the bitmap of u. Then we set $v \leftarrow u$ and continue until reaching the root, where we answer $\pi(i) = i'$. See Fig. 4 for an example.

3.2.2. Query time

In both queries the time is $\mathcal{O}(\ell)$, where ℓ is the depth of the leaf arrived at. If i is chosen uniformly at random in $[1..n]$, then the average cost is $\frac{1}{n}\sum n_i \ell_i \in \mathcal{O}(1 + \mathcal{H}(\texttt{vRuns}))$. However, the worst case can be $\mathcal{O}(\texttt{nRuns})$ in a fully skewed tree. We can ensure $\ell \in \mathcal{O}(\log \texttt{nRuns})$ in the worst case while maintaining the average case by slightly rebalancing the Huffman tree [36]. Given any constant $x > 0$, the height of the Huffman tree can be bounded to at most $(1 + x)\lg \texttt{nRuns}$ so that the total number of bits added to the encoding is at most $n \cdot \texttt{nRuns}^{-x \lg \varphi}$, where $\varphi \approx 1.618$ is the golden ratio. This is $o(n)$ if $\texttt{nRuns} \in \omega(1)$, otherwise the cost is $\mathcal{O}(\texttt{nRuns}) \subset \mathcal{O}(1)$ anyway. Similarly, the average time stays $\mathcal{O}(1 + \mathcal{H}(\texttt{vRuns}))$, as it increases at most by $\mathcal{O}(\texttt{nRuns}^{-x \log \varphi}) \subset \mathcal{O}(1)$. This rebalancing takes just $\mathcal{O}(\texttt{nRuns})$ time if the frequencies are already sorted. Note also that the space required by the query is constant.

Theorem 1. *There is an encoding scheme using at most $n\mathcal{H}(\texttt{vRuns}) + \mathcal{O}(\texttt{nRuns} \log n) + o(n)$ bits to represent a permutation π over $[1..n]$ covered by $\texttt{nRuns}$ contiguous ascending runs of lengths forming the vector $\texttt{vRuns}$. It can be built within time $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \log \texttt{nRuns})$ and constant space for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average computation time is $\mathcal{O}(1 + \mathcal{H}(\texttt{vRuns}))$.*

We note that the space analysis leading to $n\mathcal{H}(\texttt{vRuns}) + o(n)$ bits works for any tree shape. We could have used a balanced tree, yet we would not achieve $\mathcal{O}(1 + \mathcal{H}(\texttt{vRuns}))$ average time. On the other hand, by using Hu–Tucker codes instead of Huffman, as in our previous work [10], we would not need the permutation ϕ and, by using compact tree representations [46], we would be able to reduce the space to $n\mathcal{H}(\texttt{vRuns}) + \mathcal{O}(\texttt{nRuns} \log \frac{n}{\texttt{nRuns}}) + o(n)$ bits. This is interesting for large values of $\texttt{nRuns}$, as it is always $n\mathcal{H}(\texttt{vRuns}) + o(n(1 + \mathcal{H}(\texttt{vRuns})))$ even if $\texttt{nRuns} \in \Theta(n)$.[3]

3.3. Mixing ascending and descending runs

We can easily extend Theorem 1 to mix ascending and descending runs.

Corollary 1. *Theorem 1 holds verbatim if π is partitioned into a sequence $\texttt{nRuns}$ contiguous monotone (i.e., ascending or descending) runs of lengths forming the vector $\texttt{vRuns}$.*

Proof. We mark in a bitmap of length $\texttt{nRuns}$ whether each run is ascending or descending, and then reverse descending runs in π, so as to obtain a new permutation π_{asc}, which is represented using Theorem 1 (some runs of π could now be merged in π_{asc}, but we force those runs to stay separate).

The values $\pi(i)$ and $\pi^{-1}(j)$ are easily computed from π_{asc}: If $\pi_{asc}^{-1}(j) = i$, we use C to determine that i is within run $\pi_{asc}(\ell..r)$, that is, $\ell = \texttt{select}_1(\texttt{rank}_1(C, i))$ and $r = \texttt{select}_1(\texttt{rank}_1(C, i) + 1) - 1$. If that run is reversed in π, then

[3] We do not follow this path because we are more interested in multiary codes (see Section 3.5) and, to the best of our knowledge, there is no efficient (i.e., $\mathcal{O}(\texttt{nRuns} \log \texttt{nRuns})$ time) algorithm for building multiary Hu–Tucker codes [32].

$\pi^{-1}(j) = \ell + r - i$, else $\pi^{-1}(j) = i$. For $\pi(i)$, we use C to determine that i belongs to run $\pi(\ell..r)$. If the run is descending, then we return $\pi_{asc}(\ell + r - i)$, else we return $\pi_{asc}(i)$. The operations on C require only constant time. The extra construction time is just $\mathcal{O}(n)$, and no extra space is needed apart from $\texttt{nRuns} \in o(\texttt{nRuns} \log n)$ bits. □

Note that, unlike the case of ascending runs, where there is an obviously optimal way of partitioning (that is, maximize the run lengths), we have some freedom when partitioning into ascending or descending runs, at the endpoints of the runs: If an ascending (resp. descending) run is followed by a descending (resp. ascending) run, the limiting element can be moved to either run; if two ascending (resp. descending) runs are consecutive, one can create a new descending (resp. ascending) run with the two endpoint elements. While finding the optimal partitioning might not be easy, we note that these decisions cannot affect more than $\mathcal{O}(\texttt{nRuns})$ elements, and thus the entropy of the partition cannot be modified by more than $\mathcal{O}(\texttt{nRuns} \log n)$, which is absorbed by the redundancy of our representation.

3.4. Improved adaptive sorting

One of the best known sorting algorithms is `MergeSort`, based on a simple procedure to merge two already sorted arrays, and with a complexity of $n\lceil \lg n \rceil$ comparisons and $\mathcal{O}(n \log n)$ running time. It had been already noted [32] that finding the down-steps of the array in linear time allows improving the time of `MergeSort` to $\mathcal{O}(n(1 + \log \texttt{nRuns}))$ (the down-step concept can be applied to general sequences, where consecutive equal values do not break runs).

We now show that the construction process of our data structure sorts the permutation and, applied on a general sequence, it achieves a refined sorting time of $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns}))) \subset \mathcal{O}(n(1 + \log \texttt{nRuns}))$ (since $\mathcal{H}(\texttt{vRuns}) \leqslant \lg \texttt{nRuns}$).

Theorem 2. *There is an algorithm sorting an array of length n covered by* `nRuns` *contiguous monotone runs of lengths forming the vector* `vRuns` *in time* $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})))$*, which is worst-case optimal in the comparison model.*

Proof. Our construction of Theorem 1 (and Corollary 1) indeed sorts π (after converting it into π') within this time, and it also works if the array is not a permutation. This is optimal because, even considering just ascending runs, there are $\frac{n!}{n_1!n_2!\ldots n_{\texttt{nRuns}}!}$ different permutations that can be covered with runs of lengths forming the vector $\texttt{vRuns} = \langle n_1, n_2, \ldots, n_{\texttt{nRuns}} \rangle$. Thus $\lg \frac{n!}{n_1!n_2!\ldots n_{\texttt{nRuns}}!}$ comparisons are necessary. Using Stirling's approximation to the factorial we have $\lg \frac{n!}{n_1!n_2!\ldots n_{\texttt{nRuns}}!} \in (n + 1/2) \lg n - \sum_i (n_i + 1/2) \lg n_i - \mathcal{O}(\log \texttt{nRuns})$. Since $\sum \lg n_i \leqslant \texttt{nRuns} \lg(n/\texttt{nRuns})$, this is $n\mathcal{H}(\texttt{vRuns}) - \mathcal{O}(\texttt{nRuns} \log(n/\texttt{nRuns})) \subset n\mathcal{H}(\texttt{vRuns}) - \mathcal{O}(n)$. The term $\Omega(n)$ is also necessary to read the input, hence implying a lower bound of $\Omega(n(1 + \mathcal{H}(\texttt{vRuns})))$.

Note, however, that our formula $\frac{n!}{n_1!n_2!\ldots n_{\texttt{nRuns}}!}$ is actually overcounting. That is, it properly counts the set of permutations that *can be* covered with `nRuns` runs of lengths `vRuns`, but it includes permutations that can also be covered with fewer runs (as two consecutive runs could be merged). Still the lower-bound argument is valid: We have proved that the lower bound applies to the union of two classes: one (1) contains (some[4]) permutations of entropy $\mathcal{H}(\texttt{vRuns})$ and the other (2) contains (some) permutations of entropy less than $\mathcal{H}(\texttt{vRuns})$. Obviously the bound does not hold for class (2) alone, as we can sort it in less time. Since we can tell the class of a permutation in $\mathcal{O}(n)$ time by counting the down-steps, it follows that the bound also applies to class (1) alone (otherwise $\mathcal{O}(n) + o(n\mathcal{H}(\texttt{vRuns}))$ would be achievable for (1) + (2)). □

3.5. Boosting time performance

The time achieved in Theorem 1 (and Corollary 1) can be boosted by an $\mathcal{O}(\log \log n)$ time factor by using Huffman codes of higher arity. Given the run lengths `vRuns`, we build a t-ary Huffman tree for `vRuns`, with $t = \sqrt{\lg n}$. Since now we merge t children to build the parent, the sequence stored in the parent to indicate the child each element comes from is not binary, but over $[1..t]$.

The total length of all the sequences stored at all the Huffman tree nodes is $< n(1 + \mathcal{H}(\texttt{vRuns})/\lg t)$ [28]. To reduce the redundancy, we represent each sequence $S[1..m]$ stored at a node using the compressed representation of Golynski et al. [23, Lemma 9], which takes $m\mathcal{H}_0(S) + \mathcal{O}(m \log t \log \log m / \log^2 m)$ bits.

For the string $S[1..m]$ corresponding to a leaf covering runs of lengths $m_1, \ldots, m_t$, we have $m\mathcal{H}_0(S) = \sum m_i \lg \frac{m}{m_i}$. From there we can carry out exactly the same analysis done in Section 3.1 for binary trees, to conclude that the sum of the $m\mathcal{H}_0(S)$ bits for all the strings S over all the tree nodes is $n\mathcal{H}(\texttt{vRuns})$. On the other hand, the redundancies add up to $\mathcal{O}(n(1 + \mathcal{H}(\texttt{vRuns})/\log t) \log t \log \log n / \log^2 n) \subset o(n)$ bits.

The advantage of the t-ary representation is that the average leaf depth is $1 + \mathcal{H}(\texttt{vRuns})/\lg t \in \mathcal{O}(1 + \mathcal{H}(\texttt{vRuns})/\log \log n)$. The algorithms to compute $\pi(i)$ and $\pi^{-1}(i)$ are similar, except that `rank` and `select` are carried out on sequences S over alphabets of size $\sqrt{\lg n}$. Those operations can still be carried out in constant time on the representation we have chosen [23].

[4] Other permutations with vectors distinct from `vRuns` could also have entropy $\mathcal{H}(\texttt{vRuns})$.

For the worst case, if $\mathtt{nRuns} \in \omega(1)$, we can again limit the depth of the Huffman tree to $\mathcal{O}(\log \mathtt{nRuns}/\log\log n)$ and maintain the same average time. The multiary case is far less understood than the binary case. An algorithm to find the optimal length-restricted t-ary code was presented whose running time is linear once the lengths are sorted [4]. To analyze the increase in redundancy, consider the suboptimal method that simply takes any node v of depth more than $\ell = 4\lg \mathtt{nRuns}/\lg t$ and balances its subtree (so that height $5\lg \mathtt{nRuns}/\lg t$ is guaranteed). Since any node at depth ℓ covers a total length of at most $n/t^{\lfloor \ell/2 \rfloor}$ (see next paragraph), the sum of all the lengths covered by these nodes is at most $\mathtt{nRuns} \cdot n/t^{\lfloor \ell/2 \rfloor}$. By forcing those subtrees to be balanced, the average leaf depth increases by at most $(\lg \mathtt{nRuns}/\lg t)\mathtt{nRuns}/t^{\lfloor \ell/2 \rfloor} \leqslant \lg(\mathtt{nRuns})/(\mathtt{nRuns}\lg t) \in \mathcal{O}(1)$. Hence the worst case is limited to $\mathcal{O}(1 + \log \mathtt{nRuns}/\log\log n)$ while the average case stays within $\mathcal{O}(1 + \mathcal{H}(\mathtt{vRuns})/\log\log n)$. For the space, since $\mathtt{nRuns} \in \omega(1)$, we can just charge the $\lg \mathtt{nRuns}/\lg t$ levels added to all the nodes deeper than ℓ, which cover at most $\mathtt{nRuns} \cdot n/t^{\lfloor \ell/2 \rfloor}$ cells, and get $\lg \mathtt{nRuns} \cdot \mathtt{nRuns} \cdot n/t^{\lfloor \ell/2 \rfloor} = n \cdot \lg(\mathtt{nRuns})/\mathtt{nRuns} \in o(n)$ further bits.

The upper bound of $n/t^{\lfloor \ell/2 \rfloor}$ is obtained as follows. Consider a node v in the t-ary Huffman tree. Then $\mathtt{len}(u) \geqslant \mathtt{len}(v)$ for any uncle u of v, as otherwise switching v and u improves the already optimal Huffman tree (recall the definition of the covered area $\mathtt{len}(\cdot)$ from Section 3.1). Hence w, the grandparent of v (i.e., the parent of u) must cover an area of size $\mathtt{len}(w) \geqslant t \cdot \mathtt{len}(v)$. Thus the covered length is multiplied at least by t when moving from a node to its grandparent. Conversely, it is divided at least by t as we move from a node to any grandchild. As the total length at the root is n, the length covered by any node v at depth ℓ is at most $\mathtt{len}(v) \leqslant n/t^{\lfloor \ell/2 \rfloor}$.

This yields our final result for contiguous monotone runs.

Theorem 3. *There is an encoding scheme using at most $n\mathcal{H}(\mathtt{vRuns}) + \mathcal{O}(\mathtt{nRuns}\log n) + o(n)$ bits to encode a permutation π over $[1..n]$ covered by $\mathtt{nRuns}$ contiguous monotone runs of lengths forming the vector $\mathtt{vRuns}$. It can be built within time $\mathcal{O}(n(1 + \mathcal{H}(\mathtt{vRuns})/\log\log n))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1 + \log \mathtt{nRuns}/\log\log n)$ and constant space for any value of $i \in [1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average computation time is $\mathcal{O}(1 + \mathcal{H}(\mathtt{vRuns})/\log\log n)$.*

The only missing part is the construction time, since now we have to build strings $S[1..m]$ by merging t increasing runs. This can be done in $\mathcal{O}(m)$ time by using atomic heaps [19]. The compressed sequence representations are built in linear time [23]. Note that this implies that we can sort an array with $\mathtt{nRuns}$ contiguous monotone runs of lengths forming the vector $\mathtt{vRuns}$ in time $\mathcal{O}(n(1 + \mathcal{H}(\mathtt{vRuns})/\log\log n))$, yet we are not anymore in the comparison model.

This data structure yields almost directly a new representation of sequences, described in Section 6.3.

4. Strict runs

Some classes of permutations can be covered by a small number of runs of a stricter type. We present an encoding scheme that takes advantage of them.

Definition 3. A *strict ascending run* in a permutation π is a maximal range of positions satisfying $\pi(i + k) = \pi(i) + k$. The *head* of such run is its first position. The number of strict ascending runs of π is denoted by $\mathtt{nSRuns}$, and the sequence of the lengths of the strict ascending runs is denoted by $\mathtt{vSRuns}$. We will call $\mathtt{vHRuns}$ the sequence of contiguous monotone run lengths of the sequence formed by the strict run heads of π. Similarly, the notion of a *strict descending run* can be defined, as well as that of *strict* (*monotone*) *run* encompassing both.

For example, our permutation $\pi = (8, 9, 1, 4, 5, 6, 7, 2, 3)$ has $\mathtt{nSRuns} = 4$ strict runs of lengths forming the vector $\mathtt{vSRuns} = \langle 2, 1, 4, 2\rangle$. The run heads are $\langle 8, 1, 4, 2\rangle$, which form 3 monotone runs, of lengths forming the vector $\mathtt{vHRuns} = \langle 1, 2, 1\rangle$. The number of strict runs can be anywhere between $\mathtt{nRuns}$ and n; for instance the permutation $(6, 7, 8, 9, 10, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$ contains $\mathtt{nSRuns} = \mathtt{nRuns} = 2$ runs, both of which are strict, while the permutation $(1, 3, 5, 7, 9, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\mathtt{nSRuns} = 10$ strict runs, each of length 1, but only 2 runs, each of length 5.

Theorem 4. *Assume there is an encoding P for a permutation over $[1..n]$ with $\mathtt{nRuns}$ contiguous monotone runs of lengths forming the vector $\mathtt{vRuns}$, which requires $s(n, \mathtt{nRuns}, \mathtt{vRuns})$ bits of space and can apply the permutation and its inverse in time $t(n, \mathtt{nRuns}, \mathtt{vRuns})$. Now consider a permutation π over $[1..n]$ covered by $\mathtt{nSRuns}$ strict runs and by $\mathtt{nRuns} \leqslant \mathtt{nSRuns}$ monotone runs, and let $\mathtt{vHRuns}$ be the vector formed by the $\mathtt{nRuns}$ monotone run lengths in the permutation of strict run heads. Then there is an encoding scheme using at most $s(\mathtt{nSRuns}, \mathtt{nRuns}, \mathtt{vHRuns}) + \mathcal{O}(\mathtt{nSRuns}\log\frac{n}{\mathtt{nSRuns}}) + o(n)$ bits for π. It can be computed in $\mathcal{O}(n)$ time on top of that for building P. It supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(t(\mathtt{nSRuns}, \mathtt{nRuns}, \mathtt{vHRuns}))$ for any value $i \in [1..n]$.*

Proof. We first set up a bitmap R of length n marking with a 1 bit the beginning of the strict runs. We set up a second bitmap R^{inv} such that $R^{\mathrm{inv}}[i] = R[\pi^{-1}(i)]$. Now we create a new permutation π' over $[1..\mathtt{nSRuns}]$ that collapses the strict runs of π, $\pi'(i) = \mathtt{rank}_1(R^{\mathrm{inv}}, \pi(\mathtt{select}_1(R, i)))$. All this takes $\mathcal{O}(n)$ time and the bitmaps take $2\mathtt{nSRuns}\lg\frac{n}{\mathtt{nSRuns}} + \mathcal{O}(\mathtt{nSRuns}) + o(n)$ bits in compressed form [43], where $\mathtt{rank}$ and $\mathtt{select}$ are supported in constant time.

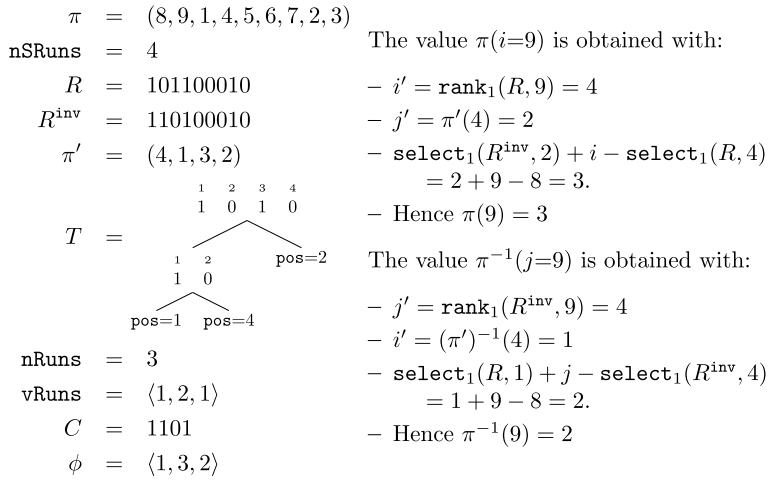


Fig. 5. Our strict runs-compressed data structure, on the permutation of Fig. 2.

Now we build the structure P for π'. The number of monotone runs in π is the same as for the sequence of strict run heads in π, and in turn the same as the runs in π'. So the number of runs in π' is also $\texttt{nRuns}$ and their lengths are $\texttt{vHRuns}$. Thus we require $s(\texttt{nSRuns}, \texttt{nRuns}, \texttt{vHRuns})$ further bits.

To compute $\pi(i)$, we find $i' \leftarrow \texttt{rank}_1(R, i)$ and then compute $j' \leftarrow \pi'(i')$. The final answer is $\texttt{select}_1(R^{\texttt{inv}}, j') + i - \texttt{select}_1(R, i')$. To compute $\pi^{-1}(j)$, we find $j' \leftarrow \texttt{rank}_1(R^{\texttt{inv}}, j)$ and then compute $i' \leftarrow (\pi')^{-1}(j')$. The final answer is $\texttt{select}_1(R, i') + j - \texttt{select}_1(R^{\texttt{inv}}, j')$. The structure requires only constant time on top of that to support the operator $\pi'()$ and its inverse $\pi'^{-1}()$. □

The theorem can be combined with previous results, for example Theorem 3, in order to obtain concrete data structures. Fig. 5 illustrates such a construction on our example permutation.

This representation is interesting because its space could be much less than n if $\texttt{nSRuns}$ is small enough. However, it still retains an $o(n)$ term that can be dominant. The following corollary describes a compressed data structure where the $o(n)$ term is significantly reduced.

Corollary 2. *The $o(n)$ term in the space of Theorem 4 can be replaced by $\mathcal{O}(\texttt{nSRuns} \log\log \frac{n}{\texttt{nSRuns}} + \log n)$ at the cost of $\mathcal{O}(1 + \log \texttt{nSRuns})$ extra time for the queries.*

Proof. Replace the structure of Raman et al. [43] by the binary searchable gap encoding of Gupta et al. [27], which takes $\mathcal{O}(1 + \log \texttt{nSRuns})$ time for $\texttt{rank}$ and $\texttt{select}$ (recall Section 2.3). □

Other trade-offs for the bitmap encodings are possible, such as the one described by Gupta [26, Theorem 18, p. 155].

5. Shuffled sequences

Up to now our runs have been contiguous in π. Levcopoulos and Petersson [33] introduced the more sophisticated concept of partitions formed by interleaved runs, such as *Shuffled UpSequences* (SUS) and *Shuffled Monotone Sequences* (SMS). We now show how to take advantage of permutations formed by shuffling (interleaving) a small number of runs.

Definition 4. A decomposition of a permutation π over $[1..n]$ into *Shuffled UpSequences* is a set of, not necessarily consecutive, disjoint subsequences of increasing numbers that cover π. The number of shuffled upsequences in such a decomposition of π is denoted by $\texttt{nSUS}$, and the vector formed by the lengths of the involved shuffled upsequences, in arbitrary order, is denoted by $\texttt{vSUS}$. When the subsequences can be of increasing or decreasing numbers, we call them *Shuffled Monotone Sequences*, call $\texttt{nSMS}$ their number and $\texttt{vSMS}$ the vector formed by their lengths.

For example, the permutation $(1, \mathbf{6}, 2, \mathbf{7}, 3, \mathbf{8}, 4, \mathbf{9}, 5, \mathbf{10})$ contains $\texttt{nSUS} = 2$ shuffled upsequences of lengths forming the vector $\texttt{vSUS} = \langle 5, 5\rangle$, but $\texttt{nRuns} = 5$ runs, all of length 2. Interestingly, we can reduce the problem of representing shuffled sequences to that of representing strings and contiguous runs.

$$\begin{array}{rcl} \pi & = & (\ 8,\ 9,\ \mathbf{1},\ \mathbf{4},\ 2,\ \mathbf{5},\ \mathbf{6},\ 3,\ \mathbf{7}\ ) \\ S & = & \quad 1\ \ 1\ \ 2\ \ 2\ \ 3\ \ 2\ \ 2\ \ 3\ \ 2 \\ A & = & 0, 2, 7, 9 \\ \pi' & = & (\ 8,\ 9,\ 1,\ 4,\ 5,\ 6,\ 7,\ 2,\ 3\ ) \end{array}$$

The value $\pi(9)$ is $\pi'(A[S[9]] + \mathtt{rank}_{S[9]}(S, 9))$, where $S[9] = 2$, $A[S[9]] = 2$, $\mathtt{rank}_2(S, 9) = 5$, hence $\pi(9) = \pi'(2 + 5) = \pi'(7) = 7$. To compute $\pi^{-1}(3)$ we start with $(\pi')^{-1}(3) = 9$, then $\ell = 3$ because $A[3] < 9 \leq A[4]$. Hence $\pi^{-1}(3) = \mathtt{select}_3(S, 9 - A[3]) = 8$.

Fig. 6. Example of an SUS-compressed data structure on a permutation that reduces to that of Fig. 2 via Theorem 5.

5.1. Reduction to strings and contiguous monotone sequences

We first show how a permutation with a small number of shuffled monotone sequences can be represented using strings over a small alphabet and permutations with a small number of contiguous monotone sequences.

Theorem 5. *Assume there exists an encoding P for a permutation over $[1..n]$ with $\mathtt{nRuns}$ contiguous monotone runs of lengths forming the vector $\mathtt{vRuns}$, which requires $s(n, \mathtt{nRuns}, \mathtt{vRuns})$ bits of space and supports the application of the permutation and its inverse in time $t(n, \mathtt{nRuns}, \mathtt{vRuns})$. Assume also that there is a data structure S for a string $S[1..n]$ over an alphabet of size $\mathtt{nSMS}$ with symbol frequencies $\mathtt{vSMS}$, using $s'(n, \mathtt{nSMS}, \mathtt{vSMS})$ bits of space and supporting operators $\mathtt{string\_rank}$, $\mathtt{string\_select}$, and access to values $S[i]$, in time $t'(n, \mathtt{nSMS}, \mathtt{vSMS})$. Now consider a permutation π over $[1..n]$ covered by $\mathtt{nSMS}$ shuffled monotone sequences of lengths $\mathtt{vSMS}$. Then there exists an encoding of π using at most $s(n, \mathtt{nSMS}, \mathtt{vSMS}) + s'(n, \mathtt{nSMS}, \mathtt{vSMS}) + \mathcal{O}(\mathtt{nSMS} \log \frac{n}{\mathtt{nSMS}}) + o(n)$ bits. Given the covering by SMSs, the encoding can be built in time $\mathcal{O}(n)$, in addition to that of building P and S. It supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $t(n, \mathtt{nSMS}, \mathtt{vSMS}) + t'(n, \mathtt{nSMS}, \mathtt{vSMS})$ for any value of $i \in [1..n]$. The result is also valid for shuffled upsequences, in which case P is just required to handle ascending runs.*

Proof. Given the partition of π into $\mathtt{nSMS}$ monotone subsequences, we create a string $S[1..n]$ over alphabet $[1..\mathtt{nSMS}]$ that indicates, for each element of π, the label of the monotone sequence it belongs to. We encode $S[1..n]$ using the data structure S. We also store an array $A[1..\mathtt{nSMS}]$ so that $A[\ell]$ is the accumulated length of all the sequences with label less than ℓ.

Now consider the permutation π' formed by the sequences taken in label order: π' can be covered with $\mathtt{nSMS}$ contiguous monotone runs $\mathtt{vSMS}$, and hence can be encoded using $s(n, \mathtt{nSMS}, \mathtt{vSMS})$ bits using P. This computes $\pi'()$ and $\pi'^{-1}()$ in time $t(n, \mathtt{nSMS}, \mathtt{vSMS})$ (again, some of the runs could be merged in π', but we avoid that). Thus $\pi(i) = \pi'(A[S[i]] + \mathtt{string\_rank}_{S[i]}(S, i))$ is computed in time $t(n, \mathtt{nSMS}, \mathtt{vSMS}) + t'(n, \mathtt{nSMS}, \mathtt{vSMS})$. Similarly, $\pi^{-1}(j) = \mathtt{string\_select}_\ell(S, (\pi')^{-1}(j) - A[\ell])$, where ℓ is such that $A[\ell] < (\pi')^{-1}(j) \leqslant A[\ell + 1]$, can also be computed in time $t(n, \mathtt{nSMS}, \mathtt{vSMS}) + t'(n, \mathtt{nSMS}, \mathtt{vSMS})$, plus the time to find ℓ. The latter is reduced to constant by representing A with a bitmap $A'[1..n]$ with the bits set at the values $A[\ell] + 1$, so that $A[\ell] = \mathtt{select}_1(A', \ell) - 1$, and then ℓ is simply computed as $\ell = \mathtt{rank}_1(A', (\pi')^{-1}(j))$. With the structure of Raman et al. [43], A' uses $\mathcal{O}(\mathtt{nSMS} \log \frac{n}{\mathtt{nSMS}}) + o(n)$ bits and operates in constant time. □

See Fig. 6 for an example of this theorem. We will now obtain concrete results by using specific representations for P and S, and specific methods to find the decomposition into shuffled sequences.

5.2. Shuffled upsequences

Given an arbitrary permutation, one can decompose it in linear time into contiguous runs in order to minimize $\mathcal{H}(\mathtt{vRuns})$, where $\mathtt{vRuns}$ is the vector of run lengths. However, decomposing the same permutation into shuffled up (resp. monotone) sequences so as to minimize either $\mathtt{nSUS}$ or $\mathcal{H}(\mathtt{vSUS})$ (resp. $\mathtt{nSMS}$ or $\mathcal{H}(\mathtt{vSMS})$) is computationally harder.

Fredman [20] gave an algorithm to compute a partition of minimum size $\mathtt{nSUS}$, into upsequences, claiming a worst case complexity of $\mathcal{O}(n \log n)$. Even though he did not claim it at the time, it is easy to observe that his algorithm is adaptive in $\mathtt{nSUS}$ and takes $\mathcal{O}(n(1 + \log \mathtt{nSUS}))$ time. We give here an improvement of his algorithm that computes the partition in time $\mathcal{O}(n(1 + \mathcal{H}(\mathtt{vSUS})))$, no worse than the time of his original algorithm since $\mathcal{H}(\mathtt{vSUS}) \leqslant \lg \mathtt{nSUS}$.

Theorem 6. *Let an array $D[1..n]$ be optimally covered by $\mathtt{nSUS}$ shuffled upsequences (equal values do not break an upsequence). Then there is an algorithm finding a covering of size $\mathtt{nSUS}$ in time $\mathcal{O}(n(1 + \mathcal{H}(\mathtt{vSUS}))) \subset \mathcal{O}(n(1 + \log \mathtt{nSUS}))$, where $\mathtt{vSUS}$ is the vector formed by the lengths of the upsequences found.*

Proof. Initialize a sequence $S_1 = (D[1])$, and a splay tree T [47] with the node (S_1), ordered by the rightmost value of the sequence contained by each node. For each further array element $D[i]$, search for the sequence with the maximum ending point no larger than $D[i]$. If it exists, add $D[i]$ to this sequence, otherwise create a new sequence and add it to T.

Fredman [20] already proved that this algorithm finds a partition of minimum size nSUS. Note that, although the rightmost values of the splay tree nodes change when we insert a new element in their sequence, their relative position with respect to the other nodes remains the same, since all the nodes at the right hold larger values than the one inserted. This implies in particular that only searches and insertions are performed in the splay tree.

A simple analysis, valid for both the plain sorted array in Fredman's proof and the splay tree of our own proof, yields an adaptive complexity of $\mathcal{O}(n(1+\log \mathrm{nSUS}))$ comparisons, since both structures contain at most nSUS elements at any time. The additional linear term (relevant when $\mathrm{nSUS}=1$) corresponds to the cost of reading each element once.

The analysis of the algorithm using the splay tree refines the complexity to $\mathcal{O}(n(1+\mathcal{H}(\mathrm{vSUS})))$, where vSUS is the vector formed by the lengths of the upsequences found. These lengths correspond to the frequencies of access to each node of the splay tree, which yields the total access time of $\mathcal{O}(n(1+\mathcal{H}(\mathrm{vSUS})))$ [47, Theorem 2]. □

The theorem obviously applies to the particular case where the array is a permutation. For permutations and, in general, integer arrays over a universe $[1..m]$, we can deviate from the comparison model and find the partition within time $\mathcal{O}(n \log\log m)$, by using y-fast tries [48] instead of splay trees.

We can now give a concrete representation for shuffled upsequences. The complete description of the permutation requires to encode the computation of the partitioning and of the comparisons performed by the sorting algorithm. This time the encoding cost of partitioning is as important as that of merging.

Theorem 7. *Let π be a permutation over $[1..n]$ that can be optimally covered by* nSUS *shuffled upsequences, and let* vSUS *be the vector formed by the lengths of an optimal decomposition found by an algorithm. Then there is an encoding scheme for π using at most $2n\mathcal{H}(\mathrm{vSUS})+\mathcal{O}(\mathrm{nSUS}\log n)+o(n)$ bits. It can be computed in additional time $\mathcal{O}(n(1+\mathcal{H}(\mathrm{vSUS})))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1+\log \mathrm{nSUS}/\log\log n)$ for any value of $i\in[1..n]$. If i is chosen uniformly at random in $[1..n]$ the average query time is $\mathcal{O}(1+\mathcal{H}(\mathrm{vSUS})/\log\log n)$.*

Proof. Once the algorithm finds the SUS partition of optimal size nSUS, and being vSUS the corresponding vector of the sizes of the subsequences of this partition, we apply Theorem 5: For the data structure S we use Theorem 8 (see later, Section 6.3), whereas for P we use Theorem 3. Note $\mathcal{H}(\mathrm{vSUS})$ is both $\mathcal{H}_0(S)$ and $\mathcal{H}(\mathrm{vRuns})$ for permutation π'. The result follows immediately. □

One would be tempted to consider the case of a permutation π covered by nSUS upsequences that form strict runs, as a particular case. Yet, this is achieved by resorting directly to Theorem 3. The corollary extends verbatim to shuffled monotone sequences.

Corollary 3. *There is an encoding scheme using at most $n\mathcal{H}(\mathrm{vSUS})+\mathcal{O}(\mathrm{nSUS}\log n)+o(n)$ bits to encode a permutation π over $[1..n]$ optimally covered by* nSUS *shuffled upsequences, of lengths forming the vector* vSUS*, and made up of strict runs. It can be built within time $\mathcal{O}(n(1+\mathcal{H}(\mathrm{vSUS})/\log\log n))$, and supports the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1+\log \mathrm{nSUS}/\log\log n)$ for any value of $i\in[1..n]$. If i is chosen uniformly at random in $[1..n]$ then the average query time is $\mathcal{O}(1+\mathcal{H}(\mathrm{vSUS})/\log\log n)$.*

Proof. It is sufficient to represent π^{-1} using Theorem 3, since in this case π^{-1} is covered by nSUS ascending runs of lengths forming the vector vSUS: If $i_0<i_1<\cdots<i_m$ forms a strict upsequence, so that $\pi(i_t)=\pi(i_0)+t$, then calling $j_0=\pi(i_0)$ we have the ascending run $\pi^{-1}(j_0+t)=i_t$ for $0\leqslant t\leqslant m$. □

Once more, our construction translates into an improved sorting algorithm, reducing the complexity $\mathcal{O}(n(1+\log \mathrm{nSUS}))$ of the algorithm by Levcopoulos and Petersson [33].

Corollary 4. *We can sort an array of length n, optimally covered by* nSUS *shuffled upsequences, in time $\mathcal{O}(n(1+\mathcal{H}(\mathrm{vSUS})))$, where* vSUS *are the lengths of the decomposition found by the algorithm of Theorem 6.*

Proof. Our construction in Theorem 7 finds, separates, and sorts the subsequences of π, all within this time (we do not need to build string S). □

5.2.1. Open problem

Note that the algorithm of Theorem 6 finds a partition of minimum size nSUS (this is what we refer to with "optimally covered"), but that the entropy $\mathcal{H}(\mathrm{vSUS})$ of this partition is not necessarily minimal: There could be another partition, even of size larger than nSUS, with lower entropy. Our results are only in function of the entropy of the partition of minimal size nSUS found. This is unsatisfactory, as the ideal would be to speak in terms of the minimum possible $\mathcal{H}(\mathrm{vSUS})$, just as we could do for $\mathcal{H}(\mathrm{vRuns})$.

Consider for instance, for some even integer n, the permutation $(1,2,\ldots,n/2-1,n,n/2,n/2+1,\ldots,n-1)$. The algorithm of Theorem 6 yields the partition $\{(1,2,\ldots,n/2-1,n),(n/2,n/2+1,\ldots,n-1)\}$ of entropy $\mathcal{H}(\langle n/2,n/2\rangle)=n\lg 2=n$. This

is suboptimal, as the partition $\{(1,2,\ldots,n/2-1,n/2,n/2+1,\ldots,n-1),(n)\}$ is of much smaller entropy, $\mathcal{H}(\langle n-1,1\rangle) = (n-1)\lg\frac{n}{n-1} + \lg n \in \mathcal{O}(\log n)$.

On the other hand, a greedy online algorithm cannot minimize the entropy of an SUS partitioning. As an example consider the permutation $(2,3,\ldots,n/2,1,n,n/2+1,\ldots,n-1)$, for some even integer n. A greedy online algorithm that after processing a prefix of the sequence minimizes the entropy of such prefix, produces the partition $\{(1,n/2+1,\ldots,n-1),(2,3,\ldots,n/2,n)\}$, of size 2 and entropy $\mathcal{H}(\langle n/2,n/2\rangle) = n$. However, a much better partition is $\{(1,n),(2,3,\ldots,n-1)\}$, of size 2 and entropy $\mathcal{H}(\langle 2,n-2\rangle) \in \mathcal{O}(\log n)$.

We doubt that the SUS partition minimizing $\mathcal{H}(\mathtt{vSUS})$ can be found within time $\mathcal{O}(n(1+\mathcal{H}(\mathtt{vSUS})))$ or even $\mathcal{O}(n(1+\log \mathtt{nSUS}))$. Proving this right or wrong is an open challenge.

5.3. Shuffled monotone sequences

No efficient algorithm is known to compute the minimum number $\mathtt{nSMS}$ of shuffled monotone sequences composing a permutation, let alone finding a partition minimizing the entropy $\mathcal{H}(\mathtt{vSMS})$ of the lengths of the subsequences. The problem is NP-hard, by reduction from the computation of the "cochromatic" number of the graph corresponding to the permutation [31]. There exist, however, approximation algorithms. For example, Fomin et al. [18] obtain a decomposition into $O(\mathtt{nSMS})$ shuffled monotone sequences in $O(n^3)$ time.

Given any such partition into monotone subsequences, if it is of smaller entropy than the partitions considered in the previous sections, this yields an improved encoding by doing just as in Theorem 7 for SUS.

6. Impact and applications

Permutations are everywhere, so that compressing their representation helps compress many other forms of data, and supporting in reasonable time the operators on permutations yields support for other operators. From a practical viewpoint, our encodings are simple enough to be implemented. Some preliminary results on inverted indexes and compressed suffix arrays show good performance on practical data sets. As an external test, the techniques were successfully used to handle scalability problems in MPI applications [29]. We describe here a selection of examples demonstrating the impact and applicability of our results.

6.1. Natural language

Consider a natural language text tokenized into word identifiers. Its *word-based inverted index* stores for each distinct word the list of its occurrences in the tokenized text, in increasing order. This is a popular data structure for text indexing [5,49]. By regarding the concatenation of the lists of occurrences of all the words, a permutation π is obtained that is formed by ν contiguous ascending runs, where ν is the vocabulary size of the text. The lengths of those runs corresponds to the frequencies of the words in the text. Therefore our representation achieves the zero-order word-based entropy of the text, which in practice compresses the text to about 25% of its original size [11]. With $\pi(i)$ we can access any position of any inverted list, and with $\pi^{-1}(j)$ we can find the word that is at any text position j. Thus the representation contains the text and its inverted index within the space of the compressed text.

6.2. Compressed suffix arrays

Compressed suffix arrays (CSAs) are data structures for indexing general texts. A family of CSAs builds on a function called Ψ [25,45,24], which is actually a permutation. Much effort was spent in compressing Ψ to the zero- or higher-order entropy of the text while supporting direct access to it. It turns out that Ψ contains σ contiguous increasing runs, where σ is the alphabet size of the text, and that the run lengths correspond to the symbol frequencies. Thus our representation of Ψ would reach the zero-order entropy of the text. It supports not only access to Ψ but also to its inverse Ψ^{-1}, which enables so-called bidirectional indexes [44], which have several interesting properties. Furthermore, Ψ contains a number of strict ascending runs that depends on the high-order entropy of the text, and this allows compressing it further [41].

6.3. An improved sequence representation

Interestingly, the results from Section 3 yield almost directly a new representation of sequences that, compared to the state of the art [16,23], provides improved average time.

Theorem 8. *Given a string $S[1..n]$ over alphabet $[1..\sigma]$ with zero-order entropy $\mathcal{H}_0(S)$, there is an encoding for S using at most $n\mathcal{H}_0(S) + \mathcal{O}(\sigma \log n) + o(n)$ bits and answering queries $S[i]$, $\mathtt{string\_rank}_c(S,i)$ and $\mathtt{string\_select}_c(S,j)$ in time $\mathcal{O}(1 + \log\sigma/\log\log n)$ for any $c \in [1..\sigma]$, $i \in [1..n]$, and $j \in [1..n_c]$, where c is the number of occurrences of c in S. When i is chosen at random in query $S[i]$, or c is chosen with probability n_c/n in queries $\mathtt{string\_rank}_c(S,i)$ and $\mathtt{string\_select}_c(S,i)$, the average query time is $\mathcal{O}(1 + \mathcal{H}_0(S)/\log\log n)$.*

Proof. We build exactly the same t-ary Huffman tree used in Theorem 3, using the frequencies n_c instead of run lengths. The sequences at each internal node are formed so as to indicate how the symbols in the child nodes are interleaved in S. This is precisely a multiary Huffman-shaped wavelet tree [24,16], and our previous analysis shows that the space used by the tree is exactly as in Theorem 3, where now the entropy is $\mathcal{H}_0(S) = \sum_c \frac{n_c}{n} \lg \frac{n}{n_c}$. The three queries are solved by going down or up the tree and using `rank` and `select` on the sequences stored at the nodes [24,16]. Under the conditions stated for the average case, one arrives at the leaf of symbol c with probability n_c/n, and then the average case complexities follow. □

6.4. Followup

Our preliminary results [10] have stimulated further research. This is just a glimpse of the work that lies ahead on this topic.

While developing, with J. Fischer, compressed indexes for Range Minimum Query indexes based on Left-to-Right Minima (LRM) trees [17,46], we realized that LRM trees yield a technique to rearrange in linear time $\mathtt{nRuns}$ contiguous ascending runs of lengths forming vector $\mathtt{vRuns}$, into a partition of $\mathtt{nLRM} = \mathtt{nRuns}$ ascending subsequences of lengths forming a new vector $\mathtt{vLRM}$, of smaller entropy $\mathcal{H}(\mathtt{vLRM}) \leqslant \mathcal{H}(\mathtt{vRuns})$ [7]. Compared to an SUS partition, the LRM partition can have larger entropy, but it is much cheaper to compute and encode. We represent it in Fig. 1 between $\mathcal{H}(\mathtt{vRuns})$ and $\mathcal{H}(\mathtt{vSUS})$.

Barbay [6] described compressed data structures for permutations inspired in other measures of disorder and adaptive sorting algorithms than those considered in this work. One such data structure takes advantage of both the number $\mathtt{nRuns}$ and the minimum number $\mathtt{nRem}$ of elements to remove from a permutation in order to leave a sorted subsequence of it, and supports operators $\pi()$ and $\pi^{-1}()$ in time $O(\lg \mathtt{nRuns})$. Another structure takes advantage of the number $\mathtt{nInv}$ of inversions contained in the permutation and supports operators $\pi()$ and $\pi^{-1}()$ in constant time. We represent those results in Fig. 1 by round boxes around the corresponding disorder measures $\mathtt{nInv}$ and $\mathtt{nRem}$, and the disorder measures dominated by them.

While developing, with T. Gagie and Y. Nekrich, an elegant combination of previously known compressed string data structures to attain superior space/time trade-offs [8], we realized that this yields various compressed data structures for permutations π such that the times for $\pi()$ and $\pi^{-1}()$ are improved to log-logarithmic. While those results subsume our initial findings [10], the improved results now presented in Theorem 3 are incomparable with those [8], and in particular superior when the number of runs is polylogarithmic in n. In addition, our representation has less redundancy, $o(n)$ whenever $\sigma \in o(n/\log n)$, whereas the faster representation [8] requires $o(n(1 + \mathcal{H}(\mathtt{nRuns})))$ bits over the entropy.

Arroyuelo et al. [1] extended our result to range searches. The permutation is seen as a set of n points on an $n \times n$ grid, and they use approximations to SMS partitioning to separate the points into $\mathtt{nSMS}' = O(\mathtt{nSMS})$ increasing and decreasing subsequences (called "monotonic chains" in there). An additional "non-crossing" geometric property is enforced on the chains, which allows orthogonal range searches to be reduced to $O(\mathtt{nSMS})$ binary searches, so that using fractional cascading the search time is $O(\mathtt{nSMS} + \log n)$ plus the output size.

7. Discussion

7.1. Relation between space and time

Bentley and Yao [12] introduced a family of search algorithms adaptive to the position of the element sought (also known as the "unbounded search" problem) through the definition of a family of adaptive codes for unbounded integers, hence proving that the link between algorithms and encodings was not limited to the complexity lower bounds suggested by information theory. Such a relation between "time" and "space" can be found in other contexts: algorithms to merge two sets define an encoding for sets [3], and the binary results of the comparisons of any deterministic sorting algorithm in the comparison model yields an encoding of the permutation being sorted.

We have shown that some concepts originally defined for adaptive variants of the algorithm `MergeSort`, such as runs and shuffled sequences, are useful in terms of the compression of permutations, and conversely, that concepts originally defined for data compression, such as the entropy of the sets of run lengths, are a useful addition to the set of difficulty measures previously considered in the study of adaptive sorting algorithms.

More work is required to explore the application of the many other measures of preorder introduced in the study of adaptive sorting algorithms to the compression of permutations. Fig. 1 represents graphically the relation between known measures of disorder (adding to those described by Moffat and Petersson [37], those described in this and other recent work [7,6]) and a preorder on them based on optimality implications in terms of the number of comparisons performed. This is relevant for the space used by potential compressed data structures on those permutations. Yet other relations of interest should be studied, such as those in terms of optimality of the running time of the algorithm, which can be distinct from the optimality in terms of the number of comparisons performed. For instance, we saw that $\mathcal{H}(\mathtt{vSMS})$-optimality implies $\mathcal{H}(\mathtt{vSUS})$-optimality in terms of the number of comparison performed, but not in terms of the running time.

7.2. Adaptive operators

It is worth noticing that, in many cases, the time to support the operators on the compressed permutations is *smaller* as the permutation is more compressed, in opposition with the traditional setting where one needs to decompress part or all of the data in order to support the operators. This behavior, incidental in our study, is a very strong incentive to further develop the study of difficulty or compressibility measures: measures such that "easy" instances can both be compressed and manipulated in better time capture the essence of the data.

7.3. Compressed indices

Interestingly enough, our encoding techniques for permutations compress both the permutation and its index (i.e., the extra data to speed up the operators). This is opposed to previous work [39] on the encoding of permutations, whose data encoding was fixed; and to previous work [9] where the data itself can be compressed but not the index, to the point where the space used by the index dominates that used by the data itself. This direction of research is promising, as in practice it is more interesting to compress the whole succinct data structure or at least its index, rather than just the data.

Acknowledgements

We thank Ian Munro, Ola Petersson and Alistair Moffat for interesting discussions.

References

[1] D. Arroyuelo, F. Claude, R. Dorrigiv, S. Durocher, M. He, A. López-Ortiz, I. Munro, P. Nicholson, A. Salinger, M. Skala, Untangled monotonic chains and adaptive range search, Theor. Comput. Sci. 412 (32) (2011) 4200–4211.
[2] D. Arroyuelo, G. Navarro, K. Sadakane, Stronger Lempel–Ziv based compressed text indexing, Algorithmica 62 (1) (2012) 54–101.
[3] B.T. Ávila, E.S. Laber, Merge source coding, in: Proc. IEEE International Symposium on Information Theory (ISIT), 2009, pp. 214–218.
[4] M. Baer, D-ary bounded-length Huffman coding, CoRR, arXiv:cs/0701012v2, 2007.
[5] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, 2nd edition, Addison-Wesley, 2011.
[6] J. Barbay, From time to space: Fast algorithms that yield small and fast data structures, in: A. Brodnik, A. López-Ortiz, V. Raman, A. Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms (IanFest), in: LNCS, vol. 8066, Springer, 2013, pp. 97–111.
[7] J. Barbay, J. Fischer, G. Navarro, LRM-trees: Compressed indices, adaptive sorting, and compressed permutations, Theor. Comput. Sci. 459 (2012) 26–41.
[8] J. Barbay, T. Gagie, G. Navarro, Y. Nekrich, Alphabet partitioning for compressed rank/select and applications, in: Proc. 21st International Symposium on Algorithms and Computation (ISAAC), in: LNCS, vol. 6507, 2010, pp. 315–326.
[9] J. Barbay, M. He, J.I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multilabeled trees, ACM Trans. Algorithms 7 (4) (2011) 52.
[10] J. Barbay, G. Navarro, Compressed representations of permutations, and applications, in: Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS), 2009, pp. 111–122.
[11] T. Bell, J. Cleary, I. Witten, Text Compression, Prentice-Hall, 1990.
[12] J.L. Bentley, A.C.-C. Yao, An almost optimal algorithm for unbounded searching, Inf. Process. Lett. 5 (3) (1976) 82–87.
[13] N. Brisaboa, M. Luaces, G. Navarro, D. Seco, Space-efficient representations of rectangle datasets supporting orthogonal range querying, Inf. Syst. 35 (5) (2013) 635–655.
[14] Y.-F. Chien, W.-K. Hon, R. Shah, J. Vitter, Geometric Burrows–Wheeler transform: Linking range searching and text indexing, in: Proc. 18th Data Compression Conference (DCC), 2008, pp. 252–261.
[15] T. Cover, J. Thomas, Elements of Information Theory, Wiley, 1991.
[16] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Trans. Algorithms 3 (2) (2007) 20.
[17] J. Fischer, Optimal succinctness for range minimum queries, in: Proc. 9th Symposium on Latin American Theoretical Informatics (LATIN), in: LNCS, vol. 6034, 2010, pp. 158–169.
[18] F. Fomin, D. Kratsch, J. Novelli, Approximating minimum cocolorings, Inf. Process. Lett. 84 (2002) 285–290.
[19] M. Fredman, D. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, J. Comput. Syst. Sci. 48 (3) (1994) 533–551.
[20] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[21] A. Golynski, Optimal lower bounds for rank and select indexes, Theor. Comput. Sci. 387 (3) (2007) 348–359.
[22] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. 17th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 368–373.
[23] A. Golynski, R. Raman, S. Rao, On the redundancy of succinct data structures, in: Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT), in: LNCS, vol. 5124, 2008, pp. 148–159.
[24] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
[25] R. Grossi, J. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2006) 378–407.
[26] A. Gupta, Succinct data structures, Ph.D. thesis, Dept. of Computer Science, Duke University, 2007.
[27] A. Gupta, W.-K. Hon, R. Shah, J. Vitter, Compressed data structures: Dictionaries and data-aware measures, in: Proc. 16th Data Compression Conference (DCC), 2006, pp. 213–222.
[28] D. Huffman, A method for the construction of minimum-redundancy codes, Proc. IRE 40 (9) (1952) 1090–1101.
[29] H. Kamal, S. Mirtaheri, A. Wagner, Scalability of communicators and groups in MPI, in: Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010, pp. 264–275.
[30] J. Kärkkäinen, Repetition-based text indexes, Ph.D. thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999, report A-1999-4.
[31] A.E. Kézdy, H.S. Snevily, C. Wang, Partitioning permutations into increasing and decreasing subsequences, J. Comb. Theory, Ser. A 73 (2) (1996) 353–359.
[32] D.E. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching, 2nd edition, Addison-Wesley Professional, 1998.
[33] C. Levcopoulos, O. Petersson, Sorting shuffled monotone sequences, Inf. Comput. 112 (1) (1994) 37–50.
[34] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theor. Comput. Sci. 387 (3) (2007) 332–347.

[35] H. Mannila, Measures of presortedness and optimal sorting algorithms, IEEE Trans. Comput. 34 (1985) 318–325.
[36] R.L. Milidiú, E.S. Laber, Bounding the inefficiency of length-restricted prefix codes, Algorithmica 31 (4) (2001) 513–529.
[37] A. Moffat, O. Petersson, An overview of adaptive sorting, Aust. Comput. J. 24 (2) (1992) 70–77.
[38] I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), in: LNCS, vol. 1180, 1996, pp. 37–42.
[39] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations and functions, Theor. Comput. Sci. 438 (2012) 74–88.
[40] J.I. Munro, P.M. Spira, Sorting and searching in multisets, SIAM J. Comput. 5 (1) (1976) 1–8.
[41] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007) 2.
[42] M. Pătraşcu, Succincter, in: Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS), 2008, pp. 305–313.
[43] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (4) (2007) 43.
[44] L. Russo, G. Navarro, A. Oliveira, P. Morales, Approximate string matching with compressed indexes, Algorithms 2 (3) (2009) 1105–1136.
[45] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, J. Algorithms 48 (2) (2003) 294–313.
[46] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proc. 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 134–149.
[47] D. Sleator, R. Tarjan, Self-adjusting binary search trees, J. ACM 32 (3) (1985) 652–686.
[48] D. Willard, Log-logarithmic worst case range queries are possible in space $\Theta(n)$, Inf. Process. Lett. 17 (1983) 81–84.
[49] I. Witten, A. Moffat, T. Bell, Managing Gigabytes, 2nd edition, Morgan Kaufmann Publishers, 1999.