Compressed Representations of Permutations, and Applications

Jérémy Barbay, Gonzalo Navarro

Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 111–122
www.stacs-conf.org

COMPRESSED REPRESENTATIONS OF PERMUTATIONS, AND APPLICATIONS

JÉRÉMY BARBAY AND GONZALO NAVARRO

Dept. of Computer Science (DCC), University of Chile.
E-mail address: {jbarbay,gnavarro}@dcc.uchile.cl

Abstract. We explore various techniques to compress a permutation π over n integers, taking advantage of ordered subsequences in π, while supporting its application $\pi(i)$ and the application of its inverse $\pi^{-1}(i)$ in small time. Our compression schemes yield several interesting byproducts, in many cases matching, improving or extending the best existing results on applications such as the encoding of a permutation in order to support iterated applications $\pi^k(i)$ of it, of integer functions, and of inverted lists and suffix arrays.

1. Introduction

Permutations of the integers $[n] = \{1, \ldots, n\}$ are a basic building block for the succinct encoding of integer functions [38], strings [1, 18, 39, 41], and binary relations [5, 4], among others. A permutation π is trivially representable in $n\lceil \lg n \rceil$ bits, which is within $\mathcal{O}(n)$ bits of the information theory lower bound of $\lg(n!)$ bits.[1] In many interesting applications, efficient computation of both the permutation $\pi(i)$ and its inverse $\pi^{-1}(i)$ is required.

The lower bound of $\lg(n!)$ bits yields a lower bound of $\Omega(n \log n)$ comparisons to sort such a permutation in the comparison model. Yet, a large body of research has been dedicated to finding better sorting algorithms which can take advantage of specificities of each permutation to sort. Trivial examples are permutations sorted such as the identity, or containing sorted blocks [32] (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 containing sorted subsequences [28] (e.g. $(1, \mathbf{6}, 2, \mathbf{7}, 3, \mathbf{8}, 4, \mathbf{9}, 5, \mathbf{10})$): algorithms performing only $\mathcal{O}(n)$ comparisons on such permutations, yet still $\mathcal{O}(n \log n)$ comparisons in the worst case, are achievable and obviously preferable. Less trivial examples are classes of permutations whose structure makes them interesting for applications: see Mannila's seminal paper [32] and Estivil-Castro and Wood's review [14] for more details.

Each sorting algorithm in the comparison model yields an encoding scheme for permutations: It suffices to note the result of each comparison performed to uniquely identify the permutation sorted, and hence to encode it. Since an adaptive sorting algorithm performs $o(n \log n)$ comparisons on many classes of permutations, each adaptive algorithm yields a *compression scheme* for permutations, at the cost of losing a constant factor on some other

Key words and phrases: Compression, Permutations, Succinct Data Structures, Adaptive Sorting.
Second author partially funded by Fondecyt Grant 1-080019, Chile.

[1] In this paper we use the notations $\lg x = \log_2 x$ and $[x] = \{1, \ldots, x\}$.

"bad" classes of permutations. We show in Section 4 some examples of applications where only "easy" permutations arise. Yet such compression schemes do not necessarily support in reasonable time the inverse of the permutation, or even the simple application of the permutation: this is the topic of our study. We describe several encodings of permutations so that on interesting classes of instances the encoding uses $o(n \log n)$ bits while supporting the operations $\pi(i)$ and $\pi^{-1}(i)$ in time $o(\log n)$. Later, we apply our compression schemes to various scenarios, such as the encoding of integer functions, text indexes, and others, yielding original compression schemes for these abstract data types.

2. Previous Work

Definition 2.1. The *entropy* of a sequence of positive integers $X = \langle n_1, n_2, \ldots, n_r \rangle$ adding up to n is $H(X) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$. By convexity of the logarithm, $\frac{r \lg n}{n} \leq H(X) \leq \lg r$.

Succinct Encodings of Sequences. Let $S[1, n]$ be a sequence over an alphabet $[r]$. This includes bitmaps when $r = 2$ (where, for convenience, the alphabet will be $\{0, 1\}$). We will make use of succinct representations of S that support operations *rank* and *select*: $rank_c(S, i)$ gives the number of occurrences of c in $S[1, i]$ and $select_c(S, j)$ gives the position in S of the jth occurrence of c.

For the case $r = 2$, S requires n bits of space and *rank* and *select* can be supported in constant time using $\mathcal{O}(\frac{n \log \log n}{\log n}) = o(n)$ bits on top of S [36, 10, 17]. The extra space is more precisely $\mathcal{O}(\frac{n \log b}{b} + 2^b \operatorname{polylog}(b))$ for some parameter b, which is chosen to be, say, $b = \frac{1}{2} \lg n$ to achieve the given bounds. In this paper, we will sometimes apply the technique over sequences of length $\ell = o(n)$ (n will be the length of the permutations). Still, we will maintain the value of b as a function of n, not ℓ, which ensures that the extra space will be of the form $\mathcal{O}(\frac{\ell \log \log n}{\log n})$, i.e., it will tend to zero when divided by ℓ as n grows, even if ℓ stays constant. All of our $o()$ terms involving several variables in this paper can be interpreted in this strong sense: asymptotic in n. Thus we will write the above space simply as $o(\ell)$.

Raman *et al.* [40] devised a bitmap representation that takes $nH_0(S) + o(n)$ bits, while maintaining the constant time for the operations. Here $H_0(S) = H(\langle n_1, n_2, \ldots, n_r \rangle) \leq \lg r$, where n_c is the number of occurrences of symbol c in S, is the so-called *zero-order entropy* of S. For the binary case this simplifies to $nH_0(S) = m \lg \frac{n}{m} + (n-m) \lg \frac{n}{n-m} = m \lg \frac{n}{m} + \mathcal{O}(m)$, where m is the number of bits set in S.

Grossi *et al.* [19] extended the result to larger alphabets using the so-called *wavelet tree*, which decomposes a sequence into several bitmaps. By representing those bitmaps in plain form, one can represent S using $n\lceil \lg r \rceil (1 + o(1))$ bits of space, and answer $S[i]$, as well as *rank* and *select* queries on S, in time $\mathcal{O}(\log r)$. By, instead, using Raman *et al.*'s representation for the bitmaps, one achieves $nH_0(S) + o(n \log r)$ bits of space, and the same times. Ferragina *et al.* [15] used multiary wavelet trees to maintain the same compressed space, while improving the times for all the operations to $\mathcal{O}(1 + \frac{\log r}{\log \log n})$.

Measures of Disorder in Permutations. Various previous studies on the presortedness in sorting considered in particular the following measures of order on an input array to be sorted. Among others, Mehlhorn [34] and Guibas *et al.* [21] considered the number of pairs in the wrong order, Knuth [27] considered the number of ascending substrings (runs), Cook and Kim [12], and later Mannila [32] considered the number of elements which have to be

removed to leave a sorted list, Mannila [32] considered the smallest number of exchanges of arbitrary elements needed to bring the input into ascending order, Skiena [44] considered the number of encroaching sequences, obtained by distributing the input elements into sorted sequences built by additions to both ends, and Levcopoulos and Petersson [28] considered Shuffled UpSequences and Shuffled Monotone Sequences. Estivil-Castro and Wood [14] list them all and some others.

3. Compression Techniques

We first introduce a compression method that takes advantage of (ascending) runs in the permutation. Then we consider a stricter variant of the runs, which allows for further compression in applications when those runs arise, and in particular allows the representation size to be sublinear in n. Next, we consider a more general type of runs, which need not be contiguous.

3.1. Wavelet Tree on Runs

One of the best known sorting algorithm is merge sort, based on a simple linear procedure to merge two already sorted arrays, resulting in a worst case complexity of $\mathcal{O}(n \log n)$. Yet, checking in linear time for *down-step* positions in the array, where an element is followed by a smaller one, partitions the original arrays into ascending runs which are already sorted. This can speed up the algorithm when the array is partially sorted [27]. We use this same observation to encode permutations.

Definition 3.1. A *down step* of a permutation π over $[n]$ is a position i such that $\pi(i+1) < \pi(i)$. A *run* in a permutation π is a maximal range of consecutive positions $\{i, \ldots, j\}$ which 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 runs of π is noted $\rho = k + 1$, and the sequence of the lengths of the runs is noted $\texttt{Runs} = \langle d_1, d_2 - d_1, \ldots, d_k - d_{k-1}, n + 1 - d_k \rangle$.

For example, permutation $(\mathit{1}, \mathit{3}, \mathit{5}, \mathit{7}, \mathit{9}, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\rho = 2$ runs, of lengths $\langle 5, 5 \rangle$. Whereas previous analyses [32] of adaptive sorting algorithms considered only the number ρ of runs, we refine them to consider the distribution Runs of the sizes of the runs.

Theorem 3.2. *There is an encoding scheme using at most* $n(2 + H(\texttt{Runs}))(1 + o(1)) + \mathcal{O}(\rho \log n)$ *bits to encode a permutation* π *over* $[n]$ *covered by* ρ *runs of lengths* Runs. *It supports* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1 + \log \rho)$ *for any value of* $i \in [n]$. *If* i *is chosen uniformly at random in* $[n]$ *then the average time is* $\mathcal{O}(1 + H(\texttt{Runs}))$.

Proof. The Hu-Tucker algorithm [23] (see also Knuth [27, p. 446]) produces in $\mathcal{O}(\rho \log \rho)$ time a prefix-free code from a sequence of frequencies $X = \langle n_1, n_2, \ldots, n_\rho \rangle$ adding up to n, so that (1) the i-th lexicographically smallest code is that for frequency n_i, and (2) if ℓ_i is the bit length of the code assigned to the i-th sequence element, then $L = \sum \ell_i n_i$ is minimal and moreover $L < n(2 + H(X))$ [27, p. 446, Eq. (27)].

We first determine Runs in $\mathcal{O}(n)$ time, and then apply the Hu-Tucker algorithm to Runs. We arrange the set of codes produced in a binary trie (equivalent to a Huffman tree [24]), where each leaf corresponds to a run and points to its two endpoints in π. Because of property (1), reading the leaves left-to-right yields the runs also in left-to-right order. Now we convert this trie into a wavelet-tree-like structure [19] without altering its shape,

as follows. Starting from the root, first process recursively each child. For the leaves do nothing. Once both children of an internal node have been processed, the invariant is that they point to the contiguous area in π covering all their leaves, and that this area of π has already been sorted. Now we merge the areas of the two children in time proportional to the new area created (which, again, is contiguous in π because of property (1)). As we do the merging, each time we take an element from the left child we append a 0 bit to a bitmap we create for the node, and a 1 bit when we take an element from the right list.

When we finish, we have the following facts: (1) π has been sorted, (2) the time for sorting has been $\mathcal{O}(n + \rho \log \rho)$ plus the total number of bits appended to all bitmaps, (3) each of the n_i elements of leaf i (at depth ℓ_i) has been merged ℓ_i times, contributing ℓ_i bits to the bitmaps of its ancestors, and thus the total number of bits is $\sum n_i \ell_i$.

Therefore, the total number of bits in the Hu-Tucker-shaped wavelet tree is at most $n(2+H(\texttt{Runs}))$. To this we must add the $\mathcal{O}(\rho \log n)$ bits of the tree pointers. We preprocess all the bitmaps for *rank* and *select* queries so as to spend $o(n(2+H(\texttt{Runs}))$ extra bits (§2).

To compute $\pi^{-1}(i)$ we start at offset i at the root bitmap B, with position $p \leftarrow 0$, and bitmap size $s \leftarrow n$. If $B[i] = 0$ we go down to the left child with $i \leftarrow rank_0(B, i)$ and $s \leftarrow rank_0(B, s)$. Otherwise we go down to the right child with $i \leftarrow rank_1(B, i)$, $p \leftarrow p + rank_0(B, s)$, and $s \leftarrow rank_1(B, s)$. When we reach a leaf, the answer is $p + i$.

To compute $\pi(i)$ we do the reverse process, but we must first determine the leaf v and offset j within v corresponding to position i: We start at the root bitmap B, with bitmap size $s \leftarrow n$ and position $j \leftarrow i$. If $rank_0(B, s) \geq j$ we go down to the left child with $s \leftarrow rank_0(B, s)$. Otherwise we go down to the right child with $j \leftarrow j - rank_0(B, s)$ and $s \leftarrow rank_1(B, s)$. We eventually reach leaf v, and the offset within v is j. We now start an upward traversal using the nodes that are already in the recursion stack (those will be limited to $\mathcal{O}(\log \rho)$ soon). If v is a left child of its parent u, then we set $j \leftarrow select_0(B, j)$, else we set $j \leftarrow select_1(B, j)$, where B is the bitmap of u. Then we set $v \leftarrow u$ until reaching the root, where $j = \pi(i)$.

In both cases the time is $\mathcal{O}(\ell)$, where ℓ is the depth of the leaf arrived at. If i is chosen uniformly at random in $[n]$, then the average cost is $\frac{1}{n}\sum n_i \ell_i = \mathcal{O}(1 + H(\texttt{Runs}))$. However, the worst case can be $\mathcal{O}(\rho)$ in a fully skewed tree. We can ensure $\ell = \mathcal{O}(\log \rho)$ in the worst case while maintaining the average case by slightly rebalancing the Hu-Tucker tree: If there exist nodes at depth $\ell = 4 \lg \rho$, we rebalance their subtrees, so as to guarantee maximum depth $5 \lg \rho$. This affects only marginally the size of the structure. A node at depth ℓ cannot add up to a frequency higher than $n/2^{\lfloor \ell/2 \rfloor} \leq 2n/\rho^2$ (see next paragraph). Added over all the possible ρ nodes we have a total frequency of $2n/\rho$. Therefore, by rebalancing those subtrees we add at most $\frac{2n \lg \rho}{\rho}$ bits. This is $o(n)$ if $\rho = \omega(1)$, and otherwise the cost was $\mathcal{O}(\rho) = \mathcal{O}(1)$ anyway. For the same reasons the average time stays $\mathcal{O}(1 + H(\texttt{Runs}))$ as it increases at most by $\mathcal{O}(\frac{\log \rho}{\rho}) = \mathcal{O}(1)$.

The bound on the frequency at depth ℓ is proved as follows. Consider the node v at depth ℓ, and its grandparent u. Then the uncle of v cannot have smaller frequency than v. Otherwise we could improve the already optimal Hu-Tucker tree by executing either a single (if v is left-left or right-right grandchild of u) or double (if v is left-right or right-left grandchild of u) AVL-like rotation that decreases the depth of v by 1 and increases that of the uncle of v by 1. Thus the overall frequency at least doubles whenever we go up two nodes from v, and this holds recursively. Thus the weight of v is at most $n/2^{\lfloor \ell/2 \rfloor}$. ∎

The general result of the theorem can be simplified when the distribution Runs is not particularly favorable.

Corollary 3.3. *There is an encoding scheme using at most* $n\lceil \lg \rho \rceil (1 + o(1)) + \mathcal{O}(\log n)$ *bits to encode a permutation* π *over* $[n]$ *with a set of* ρ *runs. It supports* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1 + \log \rho)$ *for any value of* $i \in [n]$.

As a corollary, we obtain a new proof of a well-known result on adaptive algorithms telling that one can sort in time $O(n(1 + \log \rho))$ [32], now refined to consider the entropy of the partition and not only its size.

Corollary 3.4. *We can sort an array of length* n *covered by* ρ *runs of lengths* Runs *in time* $\mathcal{O}(n(1 + H(\texttt{Runs})))$*, which is worst-case optimal in the comparison model among all permutations with* ρ *runs of lengths* Runs *so that* $\rho \log n = o(nH(\texttt{Runs}))$.

3.2. Stricter Runs

Some classes of permutations can be covered by a small number of runs of a stricter type. We present an encoding scheme which uses $o(n)$ bits for encoding the permutations from those classes, and still $\mathcal{O}(n \lg n)$ bits for all others.

Definition 3.5. A *strict 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 runs of π is noted τ, and the sequence of the lengths of the strict runs is noted SRuns. We will call HRuns the sequence of run lengths of the sequence formed by the strict run heads of π.

For example, permutation $(6, 7, 8, 9, 10, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5})$ contains $\tau = 2$ strict runs, of lengths $\texttt{SRuns} = \langle 5, 5 \rangle$. The run heads are $\langle 6, \mathbf{1} \rangle$, and contain 2 runs, of lengths $\texttt{HRuns} = \langle 1, 1 \rangle$. Instead, $(1, 3, 5, 7, 9, \mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10})$ contains $\tau = 10$ strict runs, all of length 1.

Theorem 3.6. *There is an encoding scheme using at most* $\tau H(\texttt{HRuns})(1+o(1))+2\tau \lg \frac{n}{\tau} + o(n) + \mathcal{O}(\tau + \rho \log \tau)$ *bits to encode a permutation* π *over* $[n]$ *covered by* τ *strict runs and by* $\rho \leq \tau$ *runs, and with* HRuns *being the* ρ *run lengths in the permutation of strict run heads. It supports* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1 + \log \rho)$ *for any value of* $i \in [n]$*. If* i *is chosen uniformly at random in* $[n]$ *then the average time is* $\mathcal{O}(1 + H(\texttt{HRuns}))$.

Proof. We first set up a bitmap R marking with a 1 bit the beginning of the strict runs. Set up a second bitmap R^{inv} such that $R^{inv}[i] = R[\pi^{-1}(i)]$. Now we create a new permutation π' of $[\tau]$ which collapses the strict runs of π, $\pi'(i) = rank_1(R^{inv}, \pi(select_1(R, i)))$. All this takes $\mathcal{O}(n)$ time and the bitmaps take $2\tau \lg \frac{n}{\tau} + \mathcal{O}(\tau) + o(n)$ bits using Raman *et al.*'s technique, where *rank* and *select* are solved in constant time (§2).

Now build the structure of Thm. 3.2 for π'. The number of down steps in π is the same as for the sequence of strict run heads in π, and in turn the same as the down steps in π'. So the number of runs in π' is also ρ and their lengths are HRuns. Thus we get at most $\tau(2+H(\texttt{HRuns}))(1+o(1))+\mathcal{O}(\rho \log \tau)$ bits to encode π', and can compute π' and its inverse in $\mathcal{O}(1 + \log \rho)$ worst case and $\mathcal{O}(1 + H(\texttt{HRuns}))$ average time.

To compute $\pi(i)$, we find $i' \leftarrow rank_1(R, i)$ and then compute $j' \leftarrow \pi'(i')$. The final answer is $select_1(R^{inv}, j')+i-select_1(R, i')$. To compute $\pi^{-1}(i)$, we find $i' \leftarrow rank_1(R^{inv}, i)$ and then compute $j' \leftarrow (\pi')^{-1}(i')$. The final answer is $select_1(R, j') + i - select_1(R^{inv}, i')$. This adds only constant time on top of that to compute π' and its inverse. ∎

Once again, we might simplify the results when the distribution HRuns is not particularly favorable, and we also obtain interesting algorithmic results on sorting.

Corollary 3.7. *There is an encoding scheme using at most* $\tau\lceil\lg\rho\rceil(1+o(1)) + 2\tau\lg\frac{n}{\tau} + \mathcal{O}(\tau) + o(n)$ *bits to encode a permutation* π *over* $[n]$ *covered by* τ *strict runs and by* $\rho \leq \tau$ *runs. It supports* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1+\log\rho)$ *for any value of* $i \in [n]$.

Corollary 3.8. *We can sort a permutation of* $[n]$*, covered by* τ *strict runs and by* ρ *runs, and* HRuns *being the run lengths of the strict run heads, in time* $\mathcal{O}(n + \tau H(\texttt{HRuns})) = \mathcal{O}(n+\tau\log\rho)$*, which is worst-case optimal, in the comparison model, among all permutations sharing these* ρ*,* τ*, and* HRuns *values, such that* $\rho\log\tau = o(\tau H(\texttt{HRuns}))$.

3.3. Shuffled Sequences

Levcopoulos and Petersson [28] introduced the more sophisticated concept of partitions formed by interleaved runs, such as *Shuffled UpSequences* (SUS). We discuss here the advantage of considering permutations formed by shuffling a small number of runs.

Definition 3.9. A decomposition of a permutation π over $[n]$ into *Shuffled UpSequences* is a set of, not necessarily consecutive, subsequences of increasing numbers that have to be removed from π in order to reduce it to the empty sequence. The minimum number of shuffled upsequences in such a decomposition of π is noted σ, and the sequence of the lengths of the involved shuffled upsequences, in arbitrary order, is noted SUS.

For example, permutation $(\mathit{1}, \mathbf{6}, \mathit{2}, \mathbf{7}, \mathit{3}, \mathbf{8}, \mathit{4}, \mathbf{9}, \mathit{5}, \mathbf{10})$ contains $\sigma = 2$ shuffled upsequences of lengths $\texttt{SUS} = \langle 5, 5\rangle$, but $\rho = 5$ runs, all of length 2. Whereas the decomposition of a permutation into runs or strict runs can be computed in linear time, the decomposition into shuffled upsequences requires a bit more time. Fredman [16] gave an algorithm to compute the size of an optimal partition, claiming a worst case complexity of $\mathcal{O}(n\log n)$. In fact his algorithm is adaptive and takes $\mathcal{O}(n(1+\log\sigma))$ time. We give here a variant of his algorithm which computes the partition itself within the same complexity, and we achieve even better time on favorable sequences SUS.

Lemma 3.10. *Given a permutation* π *over* $[n]$ *covered by* σ *shuffled upsequences of lengths* SUS*, there is an algorithm finding such a partition in time* $\mathcal{O}(n(1+H(\texttt{SUS})))$.

Proof. Initialize a sequence $S_1 = (\pi(1))$, and a splay tree T [45] with the node (S_1), ordered by the rightmost value of the sequence contained by each node. For each further element $\pi(i)$, search for the sequence with the maximum ending point smaller than $\pi(i)$. If any, add $\pi(i)$ to this sequence, otherwise create a new sequence and add it to T. Fredman [16] already proved that this algorithm computes an optimal partition. The adaptive complexity results from the mere observation that the splay tree (a simple sorted array in Fredman's proof) contains at most σ elements, and that the node corresponding to a subsequence is accessed once per element in it. Hence the total access time is $\mathcal{O}(n(1+H(\texttt{SUS})))$ [45, Thm. 2]. ∎

The complete description of the permutation requires to encode the computation of both the partitioning algorithm and the sorting one, and this time the encoding cost of partitioning is as important as that of merging.

Theorem 3.11. *There is an encoding scheme using at most* $2n(1+H(\texttt{SUS}))+o(n\log\sigma)+\mathcal{O}(\sigma\log n)$ *bits to encode a permutation* π *over* $[n]$ *covered by* σ *shuffled upsequences of lengths* SUS*. It supports the operations* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1+\log\sigma)$ *for any value of* $i\in[n]$*. If* i *is chosen uniformly at random in* $[n]$ *the average time is* $\mathcal{O}(1+H(\texttt{SUS})+\frac{\log\sigma}{\log\log n})$.

Proof. Partition the permutation π into σ shuffled upsequences using Lemma 3.10, resulting in a string S of length n over alphabet $[\sigma]$ which indicates for each element of the permutation π the label of the upsequence it belongs to. Encode S with a wavelet tree using Raman *et al.*'s compression for the bitmaps, so as to achieve $nH(\texttt{SUS})+o(n\log\sigma)$ bits of space and support retrieval of any $S[i]$, as well as symbol *rank* and *select* on S, in time $\mathcal{O}(1+\log\sigma)$ (§2). Store also an array $A[1,\sigma]$ so that $A[\ell]$ is the accumulated length of all the upsequences with label less than ℓ. Array A requires $\mathcal{O}(\sigma\log n)$ bits. Finally, consider the permutation π' formed by the upsequences taken in label order: π' has at most σ runs and hence can be encoded using $n(2+H(\texttt{SUS}))(1+o(1))+\mathcal{O}(\sigma\log n)$ bits using Thm. 3.2, as SUS in π corresponds to Runs in π'. This supports $\pi'(i)$ and $\pi'^{-1}(i)$ in time $\mathcal{O}(1+\log\sigma)$.

Now $\pi(i)=\pi'(A[S[i]]+rank_{S[i]}(S,i))$ can be computed in time $\mathcal{O}(1+\log\sigma)$. Similarly, $\pi^{-1}(i)=select_\ell(S,(\pi')^{-1}(i)-A[\ell])$, where ℓ is such that $A[\ell]<(\pi')^{-1}(i)\le A[\ell+1]$, can also be computed in $\mathcal{O}(1+\log\sigma)$ time. Thus the whole structure uses $2n(1+H(\texttt{SUS}))+o(n\log\sigma)+\mathcal{O}(\sigma\log n)$ bits and supports $\pi(i)$ and $\pi^{-1}(i)$ in time $\mathcal{O}(1+\log\sigma)$.

The obstacles to achieve the claimed average time are the operations on the wavelet tree of S, and the binary search in A. The former can be reduced to $\mathcal{O}(1+\frac{\log\sigma}{\log\log n})$ by using the improved wavelet tree representation by Ferragina *et al.* (§2). The latter is reduced to constant time by representing A with a bitmap $A'[1,n]$ with the bits set at the values $A[\ell]+1$, so that $A[\ell]=select_1(A',\ell)-1$, and the binary search is replaced by $\ell=rank_1(A',(\pi')^{-1}(i))$. With Raman *et al.*'s structure (§2), A' needs $\mathcal{O}(\sigma\log\frac{n}{\sigma})$ bits and operates in constant time. ∎

Again, we might prefer a simplified result when SUS has no interesting distribution, and we also achieve an improved result on sorting, better than the known $\mathcal{O}(n(1+\log\sigma))$.

Corollary 3.12. *There is an encoding scheme using at most* $2n\lg\sigma(1+o(1))+\sigma\lg\frac{n}{\sigma}+\mathcal{O}(\sigma)$ *bits to encode a permutation* π *over* $[n]$ *covered by* σ *shuffled upsequences. It supports the operations* $\pi(i)$ *and* $\pi^{-1}(i)$ *in time* $\mathcal{O}(1+\log\sigma)$ *for any value of* $i\in[n]$.

Corollary 3.13. *We can sort an array of length* n*, covered by* σ *shuffled upsequences of lenghts* SUS*, in time* $\mathcal{O}(n(1+H(\texttt{SUS})))$*, which is worst-case optimal, in the comparison model, among all permutations decomposable into* σ *shuffled upsequences of lenghts* SUS *such that* $\sigma\log n=o(nH(\texttt{SUS}))$.

4. Applications

4.1. Inverted Indexes

Consider a full-text inverted index which gives the word positions of any word in a text. This is a popular data structure for natural language text retrieval [3, 46], as it permits for example solving phrase queries without accessing the text. For each different text word, an increasing list of its text positions is stored.

Let n be the total number of words in a text collection $T[1,n]$ and ρ the vocabulary size (i.e., number of different words). An uncompressed inverted index requires $(\rho+n)\lceil \lg n \rceil$ bits. It has been shown [31] that, by δ-encoding the differences between consecutive entries in the inverted lists, the total space reduces to $nH_0(T) + \rho\lceil \lg n \rceil$, where $H_0(T)$ is the zero-order entropy of the text if seen as a sequence of words (§2). We note that the empirical law by Heaps [22], well accepted in Information Retrieval, establishes that ρ is small: $\rho = \mathcal{O}(n^\beta)$ for some constant $0 < \beta < 1$ depending on the text type.

Several successful methods to compress natural language text take words as symbols and use zero-order encoding, and thus the size they can achieve is lower bounded by $nH_0(T)$ [35]. If we add the differentially encoded inverted index in order to be able of searching the compressed text, the total space is at least $2nH_0(T)$.

Now, the concatenation of the ρ inverted lists can be seen as a permutation of $[n]$ with ρ runs, and therefore Thm. 3.2 lets us encode it in $n(2+H_0(T))(1+o(1))+\mathcal{O}(\rho \log n)$ bits. Within the same space we can add ρ numbers telling where the runs begin, in an array $V[1,\rho]$. Now, in order to retrieve the list of the i-th word, we simply obtain $\pi(V[i]), \pi(V[i]+1), \ldots, \pi(V[i+1]-1)$, each in $\mathcal{O}(1+\log \rho)$ time. Moreover we can extract any random position from a list, which enables binary-search-based strategies for list intersection [2, 42, 13]. In addition, we can also obtain a text passage from the (inverse) permutation: To find out $T[j]$, $\pi^{-1}(j)$ gives its position in the inverted lists, and a binary search on V finds the interval $V[i] \leq \pi^{-1}(j) < V[i+1]$, to output that $T[j] = i$th word, in $\mathcal{O}(1+\log \rho)$ time.

This result is very interesting, as it constitutes a true word-based *self-index* [39] (i.e., a compressed text index that contains the text). Similar results have been recently obtained with rather different methods [9, 11]. The cleanest one is to build a wavelet tree over T with compression [15], which achieves $nH_0(T) + o(n \log \rho) + \mathcal{O}(\rho \log n)$ bits of space, and permits obtaining $T[i]$, as well as extracting the jth element of the inverted list of the ith word with $select_i(T,j)$, all in time $\mathcal{O}(1+\frac{\log \rho}{\log \log n})$.

Yet, one advantage of our approach is that the extraction of ℓ consecutive entries $\pi^{-1}([i,i'])$ takes $\mathcal{O}(\ell(1+\log \frac{\rho}{\ell}))$ time if we do the process for all the entries as a block: Start at range $[i,i']$ at the root bitmap B, with position $p \leftarrow 0$, and bitmap size $s \leftarrow n$. Go down to both left and right children: to the left with $[i,i'] \leftarrow [rank_0(B,i), rank_0(B,i')]$, same p, and $s \leftarrow rank_0(B,s)$; to the right with $[i,i'] \leftarrow [rank_1(B,i), rank_1(B,i')]$, $p \leftarrow p + rank_0(B,s)$, and $s \leftarrow rank_1(B,s)$. Stop when the range $[i,i']$ becomes empty or when we reach a leaf, in which case report all answers $p+k$, $i \leq k \leq i'$. By representing the inverted list as π^{-1}, we can extract long inverted lists faster than the existing methods.

Corollary 4.1. *There exists a representation for a text $T[1,n]$ of integers in $[1,\rho]$ (regarded as word identifiers), with zero-order entropy H_0, that takes $n(2+H_0)(1+o(1))+\mathcal{O}(\rho \log n)$ bits of space, and can retrieve the text position of the jth occurrence of the ith text word, as well as the value $T[j]$, in $\mathcal{O}(1+\log \rho)$ time. It can also retrieve any range of ℓ successive occurrences of the ith text word in time $\mathcal{O}(\ell(1+\log \frac{\rho}{\ell}))$.*

We could, instead, represent the inverted list as π, so as to extract long text passages efficiently, but the wavelet tree representation can achieve the same result. Another interesting functionality that both representations share, and which is useful for other list intersection algorithms [6, 4], is that to obtain the first entry of a list which is larger than x. This is done with *rank* and *select* on the wavelet tree representation. In our permutation representation, we can also achieve it in $\mathcal{O}(1+\log \rho)$ time by finding out the position of a number x within a given run. The algorithm is similar to those in Thm. 3.2 that descend

to a leaf while maintaining the offset within the node, except that the decision on whether to descend left or right depends on the leaf we want to arrive at and not on the bitmap content (this is actually the algorithm to compute *rank* on binary wavelet trees [39]).

Finally, we note that our inverted index data structure supports in small time all the operations required to solve conjunctive queries on binary relations.

4.2. Suffix Arrays

Suffix arrays are used to index texts that cannot be handled with inverted lists. Given a text $T[1,n]$ of n symbols over an alphabet of size ρ, the *suffix* array $A[1,n]$ is a permutation of $[n]$ so that $T[A[i],n]$ is lexicographically smaller than $T[A[i+1],n]$. As suffix arrays take much space, several compressed data structures have been developed for them [39]. One of interest for us is the *Compressed Suffix Array (CSA)* of Sadakane [41]. It builds over a permutation Ψ of $[n]$, which satisfies $A[\Psi[i]] = (A[i] \bmod n) + 1$ (and thus lets us move virtually one position forward in the text) [20]. It turns out that, using just Ψ and $\mathcal{O}(\rho \log n)$ extra bits, one can (i) *count* the number of times a pattern $P[1,m]$ occurs in T using $\mathcal{O}(m \log n)$ applications of Ψ; (ii) *locate* any such occurrence using $\mathcal{O}(s)$ applications of Ψ, by spending $\mathcal{O}(\frac{n \log n}{s})$ extra bits of space; and (iii) *extract* a text substring $T[l,r]$ using at most $s + r - l$ applications of Ψ. Hence this is another self-index, and its main burden of space is that to represent permutation Ψ.

Sadakane shows that Ψ has at most ρ runs, and gives a representation that accesses $\Psi[i]$ in constant time by using $nH_0(T) + \mathcal{O}(n \log\log \rho)$ bits of space. It was shown later [39] that the space is actually $nH_k(T) + \mathcal{O}(n \log\log \rho)$ bits, for any $k \leq \alpha \log_\rho n$ and constant $0 < \alpha < 1$. Here $H_k(T) \leq H_0(T)$ is the kth order empirical entropy of T [33].

With Thm. 3.2 we can encode Ψ using $n(2 + H_0(T))(1 + o(1)) + \mathcal{O}(\rho \log n)$ bits of space, whose extra terms aside from entropy are better than Sadakane's. Those extra terms can be very significant in practice. The price is that the time to access Ψ is $\mathcal{O}(1 + \log \rho)$ instead of constant. On the other hand, an interesting extra functionality is that to compute Ψ^{-1}, which lets us move (virtually) one position backward in T. This allows, for example, displaying the text context around an occurrence without having to spend any extra space. Still, although interesting, the result is not competitive with recent developments [15, 30].

An interesting point is that Ψ contains $\tau \leq \min(n, nH_k(T) + \rho^k)$ strict runs, for any k [29]. Therefore, Cor. 3.7 lets us represent it using $\tau \lceil \lg \rho \rceil (1+o(1)) + 2\tau \lg \frac{n}{\tau} + \mathcal{O}(\tau) + o(n)$ bits of space. For k limited as above, this is at most $nH_k(T)(\lg \rho + 2 \lg \frac{1}{H_k(T)} + \mathcal{O}(1)) + o(n \log \rho)$ bits, which is similar to the space achieved by another self-index [29, 43], yet again it is slightly superseded by its time performance.

4.3. Iterated Permutation

Munro *et al.* [37] described how to represent a permutation π as the concatenation of its cycles, completed by a bitvector of n bits coding the lengths of the cycles. As the cycle representation is itself a permutation of $[n]$, we can use any of the permutation encodings described in §3 to encode it, adding the binary vector encoding the lengths of the cycles. It is important to note that, for a specific permutation π, the difficulty to compress its cycle encoding π' is not the same as the difficulty to encode the original permutation π.

Given a permutation π with c cycles of lengths $\langle n_1, \ldots, n_c \rangle$, there are several ways to encode it as a permutation π', depending on the starting point of each cycle ($\Pi_{i \in [c]} n_i$

choices) and the order of the cycles in the encoding ($c!$ choices). As a consequence, each permutation π with c cycles of lengths $\langle n_1, \ldots, n_c \rangle$ can be encoded by any of the $\Pi_{i \in [c]} i \times n_i$ corresponding permutations.

Corollary 4.2. *Any of the encodings from Theorems 3.2, 3.6 and 3.11 can be combined with an additional cost of at most $n + o(n)$ bits to encode a permutation π over $[n]$ composed of c cycles of lengths $\langle n_1, \ldots, n_c \rangle$ to support the operation $\pi^k(i)$ for any value of $k \in \mathbb{Z}$, in time and space function of the order in the permutation encoding of the cycles of π.*

The space "wasted" by such a permutation representation of the cycles of π is $\sum \lg n_i + c \lg c$ bits. To recover some of this space, one can define a canonical cycle encoding by starting the encoding of each cycle with its smallest value, and by ordering the cycles in order of their starting point. This canonical encoding always starts with a 1 and creates at least one shuffled upsequence of length c: it can be compressed as a permutation over $[n-1]$ with at least one shuffled upsequence of length $c+1$ through Thm 3.11.

4.4. Integer Functions

Munro and Rao [38] extended the results on permutations to arbitrary functions from $[n]$ to $[n]$, and to their iterated application $f^k(i)$, the function iterated k times starting at i. Their encoding is based on the decomposition of the function into a bijective part, represented as a permutation, and an injective part, represented as a forest of trees whose roots are elements of the permutation: the summary of the concept is that an integer function is just a "hairy permutation". Combining the representation of permutations from [37] with any representation of trees supporting the level-ancestor operator and an iterator of the descendants at a given level yields a representation of an integer function f using $(1+\varepsilon) n \lg n + \mathcal{O}(1)$ bits to support $f^k(i)$ in $\mathcal{O}(1 + |f^k(i)|)$ time, for any fixed ε, integer $k \in \mathbb{Z}$ and $i \in [n]$.

Janssen *et al.* [25] defined the *degree entropy* of an ordered tree T with n nodes, having n_i nodes with i children, as $H^*(T) = H(\langle n_1, n_2, \ldots \rangle)$, and proposed a succinct data structure for T using $nH^*(T) + \mathcal{O}(n(\lg \lg n)^2 / \lg n)$ bits to encode the tree and support, among others, the level-ancestor operator. Obviously, the definition and encoding can be generalized to a forest of k trees by simply adding one node whose k children are the roots of the k trees.

Encoding the injective parts of the function using Janssen *et al.*'s [25] succinct encoding, and the bijective parts of the function using one of our permutation encodings, yields a compressed representation of any integer function which supports its application and the application of its iterated variants in small time.

Corollary 4.3. *There is a representation of a function $f : [n] \to [n]$ that uses $n(1 + \lceil \lg \rho \rceil + H^*(T)) + o(n \lg n)$ bits to support $f^k(i)$ in $\mathcal{O}(\log \rho + |f^k(i)|)$ time, for any integer k and for any $i \in [n]$, where T is the forest representing the injective part of the function, and ρ is the number of runs in the bijective part of the function.*

5. Conclusion

Bentley and Yao [8], when introducing a family of search algorithms adaptive to the position of the element searched (aka the "unbounded search" problem), did so 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.

In this paper, we have considered the relation between the difficulty measures of adaptive sorting algorithms and some measures of "entropy" for compression techniques on permutations. In particular, we have shown that some concepts originally defined for adaptive sorting algorithms, such as runs and shuffled upsequences, 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 sizes of runs, are a useful addition to the set of difficulty measures that one can consider in the study of adaptive algorithms.

It is easy to generalize our results on runs and strict runs to take advantage of permutations which are a mix of up and down runs or strict runs (e.g. $(1, 3, 5, 7, 9, \mathbf{10}, \mathbf{8}, \mathbf{6}, \mathbf{4}, \mathbf{2})$, with only a linear extra computational and/or space cost. The generalization of our results on shuffled upsequences to SMS [28], permutations containing mixes of subsequences sorted in increasing and decreasing orders (e.g. $(1, \mathbf{10}, 2, \mathbf{9}, 3, \mathbf{8}, 4, \mathbf{7}, 5, \mathbf{6})$) is sligthly more problematic, because it is NP hard to optimally decompose a permutation into such subsequences [26], but any approximation scheme [28] would yield a good encoding.

Refer to the associated technical report [7] for a longer version of this paper, in particular including all the proofs.

References

[1] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th CPM*, LNCS 4009, pages 319–330, 2006.

[2] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. 15th CPM*, LNCS 3109, pages 400–408, 2004.

[3] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 1999.

[4] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comp. Sci.*, 2007.

[5] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th SODA*, pages 680–689, 2007.

[6] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proc. 5th WEA*, LNCS 4007, pages 146–157, 2006.

[7] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. Technical Report TR/DCC-2008-18, Department of Computer Science (DCC), University of Chile, December 2008. `http://www.dcc.uchile.cl/TR/2008/TR_DCC-2008-018.pdf`.

[8] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Proc. Lett.*, 5(3):82–87, 1976.

[9] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st SIGIR*, pages 139–146, 2008.

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

[11] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.

[12] C. Cool and D. Kim. Best sorting algorithm for nearly sorted lists. *Comm. ACM*, 23:620–624, 1980.

[13] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, pages 137–148, 2007.

[14] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comp. Surv.*, 24(4):441–476, 1992.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. on Algorithms (TALG)*, 3(2):article 20, 2007.

[16] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

[17] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th ICALP*, LNCS 4051, pages 370–381, 2006.
[18] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
[19] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
[20] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. on Computing*, 35(2):378–407, 2006.
[21] L. Guibas, E. McCreight, M. Plass, and J. Roberts. A new representation of linear lists. In *Proc. 9th STOC*, pages 49–60, 1977.
[22] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, NY, 1978.
[23] T. Hu and A. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM J. of Applied Mathematics*, 21:514–532, 1971.
[24] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.
[25] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th SODA*, pages 575–584, 2007.
[26] A. E. Kézdy, H. S. Snevily, and C. Wang. Partitioning permutations into increasing and decreasing subsequences. *J. Comb. Theory Ser. A*, 73(2):353–359, 1996.
[27] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.
[28] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Inf. Comp.*, 112(1):37–50, 1994.
[29] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.
[30] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th SPIRE*, LNCS 4726, pages 214–226, 2007.
[31] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comp. Sci.*, 387(3):332–347, 2007.
[32] H. Mannila. Measures of presortedness and optimal sorting algorithms. In *IEEE Trans. Comput.*, volume 34, pages 318–325, 1985.
[33] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. of the ACM*, 48(3):407–430, 2001.
[34] K. Mehlhorn. Sorting presorted files. In *Proc. 4th GI-Conference on Theoretical Computer Science*, LNCS 67, pages 199–212, 1979.
[35] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems (TOIS)*, 18(2):113–139, 2000.
[36] I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
[37] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, LNCS 2719, pages 345–356, 2003.
[38] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. 31st ICALP*, LNCS 3142, pages 1006–1015, 2004.
[39] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
[40] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
[41] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.
[42] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, 2007.
[43] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th SPIRE*, LNCS 5280, pages 164–175, 2008.
[44] S. S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, 1988.
[45] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. of the ACM*, 32(3):652–686, 1985.
[46] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.