Title:	Succinct and Compressed Data Structures for Permutations and Integer Functions
Name:	Jérémy Barbay
Affil./Addr.	Department of Computer Science (DCC) Faculty of Physical and Mathematical Sciences University of Chile.
Keywords:	Adaptive; Compression; Permutation; Functions.
SumOriWork:	2012; Munro, Raman, Raman, Rao 2012; Barbay, Fischer, Navarro 2013; Barbay, Navarro 2013; Barbay

Succinct and Compressed Data Structures for Permutations and Integer Functions

Jérémy Barbay

Department of Computer Science (DCC)
Faculty of Physical and Mathematical Sciences
University of Chile.

Years and Authors of Summarized Original Work

2012; Munro, Raman, Raman, Rao
2012; Barbay, Fischer, Navarro
2013; Barbay, Navarro
2013; Barbay

Keywords

Adaptive; Compression; Permutation; Functions.

Problem Definition

A basic building block for compressed data structures for texts and functions is the representation of a permutation of the integers $\{1, \ldots, n\}$, denoted by $[1..n]$. A permutation π is trivially representable in $n\lceil \lg n \rceil$ bits which is within $O(n)$ bits of the information theoretic bound of $\lg(n!)$, but instances from restricted classes of permutations can be represented using much less space.

Encoding or compressing a particular permutation is useless if one cannot access its content efficiently. Given a permutation π over $[1..n]$, an integer k and an integer $i \in [1..n]$, data structures on permutations aim to support the following operators as fast as possible, using as little additional space as possible:

- $\pi(i)$: application of the permutation to i,
- $\pi^{-1}(i)$: application of the inverse permutation to i,
- $\pi^k(i)()$: $\pi()$ iteratively applied k times starting with value i (e.g. $\pi^{(2)}(i) = \pi(\pi(i))$).

Key Results

We distinguish between two types of solutions: the succinct index and two succinct data structures for permutations introduced by Munro et al. [1], and the various compressed data structures proposed later [2, 3, 4].

Succinct Data Structures

Munro et al. [1] studied the problem of succinctly representing a permutation to support operators on it quickly. They give several solutions, described below.

"Shortcut" Index supporting $\pi()$ and $\pi^{-1}()$ Given an integer parameter t, the operators $\pi()$ and $\pi^{-1}()$ can be supported by simply writing down π in an array of n words of $\lceil \lg n \rceil$ bits each, plus an auxiliary array S of at most n/t back pointers called shortcuts: In each cycle of length at least t, every t-th element has a pointer t steps back. Then, $\pi(i)$ is simply the i-th value in the primary structure, and $\pi^{-1}(i)$ is found by moving forward until a back pointer is found and then continuing to follow the cycle to the location that contains the value i.

The trick is in the encoding of the locations of the back pointers: this is done with a simple bit vector B of length n, in which a 1 indicates that a back pointer is associated with a given location. B is augmented using within $o(n)$ additional bits so that the number of 1's up to a given position and the position of the r-th 1 can be found in constant time (i.e. using the rank and select operators on binary strings [5]). This gives the location of the appropriate back pointer in the auxiliary array S. As there are back-pointer every t elements in the cycle, finding the predecessor requires $O(t)$ memory accesses.

Theorem 1. *For any strictly positive integer n and any permutation π on $[1..n]$ which can be decomposed into δ cycles of respective sizes $c_1, \ldots, c_\delta$, there is a representation of π using within $(\sum_{i\in[1..\delta]} \lceil \frac{c_i}{t} \rceil) \lg n + 2n + o(n) \subseteq \frac{n \lg n}{t} + 2n + o(n)$ bits to support the operator $\pi()$ in constant time, and the operator $\pi^{-1}()$ in time within $O(t)$.*

Interestingly enough, Munro et al. [1] did not notice that their construction is actually an index, and that the raw encoding can be replaced by any data structure supporting the operator $\pi()$, including the compressed ones later described [4].

"Cycle" Data Structure supporting $\pi^k()$ For arbitrary i and k, $\pi^k()$ is supported by writing the cycles of π together with a bit vector B marking the beginning of each cycle. Observe that the cycle representation itself is a permutation in "standard form", call it σ. The first task is to find i in the representation: it is in position $\sigma^{-1}(i)$. The segment of the representation containing i is found through the rank and select operators on B. Then $\pi^k(i)$ is determined by taking k modulo the cycle length, moving that number of steps around the cycle starting at the position of i, and applying $\sigma()$ to obtain the value to return.

Other than the support of the inverse of σ, all operators are performed in constant time, hence the asymptotic supporting time of $\pi^k()$ depends on the supporting

time in which the data structure chosen to represent σ supports the operators $\sigma()$ and $\sigma^{-1}()$. Munro et al. [1] proposed the following, using a raw encoding of σ with a shortcut index to support $\sigma^{-1}()$:

Theorem 2. *For any strictly positive integer n and any permutation π on $[1..n]$, there is a representation of π using at most $(1+\varepsilon)n\lg n + O(n)$ bits to support the operator $\pi^k()$ in time within $O(1/\varepsilon)$, for any constant ε less than 1 and for any arbitrary value of k.*

Under a restricted model of pointer machine, this technique is optimal: using $O(n)$ extra bits (i.e. $O(n/\log n)$ extra words), time within $\Omega(\log n)$ is necessary to support both $\pi()$ and $\pi^{-1}()$.

"Benes Network" Data Structure supporting $\pi^k()$ Any permutation can be implemented by a communication network composed of switches: this is called a Benes Network, and uses even less space under the RAM model than the solutions described in the previous sections. Sparsely adding pointers accelerates the support of $\pi^k()$ to time within $O(\frac{\log n}{\log\log n})$.

Theorem 3. *For any strictly positive integer n and any permutation π on $[1..n]$, there is a representation of π using at most $\lceil \lg(n!) \rceil + O(n)$ bits to support the operator $\pi^k()$ in time within $O(\log n/\log\log n)$.*

This representation uses space within an additive term within $O(n)$ of the optimal, both on average and in the worst case over all permutations over $[1..n]$.

Compressed Data Structures

Any comparison-based sorting algorithm yields an encoding for permutations, and any adaptive sorting algorithm in the comparison model yields a compression scheme for permutations. Supporting operators on such compressed permutation in less time than required to decompress the whole of it requires some more work:

Runs Barbay and Navarro [2] described how to segment a partition into `nRuns` *runs* composed of consecutive positions forming already sorted blocks, and how to merge those via a wavelet tree. This yields a data structure compressing a permutation within space optimal over all permutations with `nRuns` runs of respective sizes given by the vector `vRuns`. This data structure supports the operators $\pi()$ and $\pi^{-1}()$ in sublinear time within $O(1+\log \texttt{nRuns})$, with the average supporting time within $O(1+\mathcal{H}(\texttt{vRuns}))$ decreasing with the entropy of the partition of the permutation into runs, where the *entropy* of a sequence of positive integers $X = \langle n_1, n_2, \ldots, n_r \rangle$ adding up to n is $\mathcal{H}(X) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$.

Theorem 4. *For any strictly positive integer n and any permutation π on $[1..n]$ which can be decomposed into `nRuns` runs of respective sizes `vRuns` $= (r_1, \ldots, r_{\texttt{nRuns}})$, there is a representation of π using at most $n\mathcal{H}(\texttt{vRuns}) + O(\texttt{nRuns}\log n) + o(n)$ bits to support the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time within $O(1+\log \texttt{nRuns})$ in the worst case over $i \in [1..n]$ and in time within $O(1+\mathcal{H}(\texttt{vRuns}))$ on average when $i \in [1..n]$ is uniformly distributed. This compressed data structure can be computed in time within $O(n(1+\mathcal{H}(\texttt{vRuns})))$, which is worst-case optimal in the comparison model over all such permutations decomposed into `nRuns` runs of respective sizes given by the vector `vRuns`.*

The partitioning takes only $n-1$ comparisons, and the construction of the compressed data structure itself is an adaptive sorting algorithm improving over previous results [6, 7].

Heads of Strict Runs A two-level partition of the permutation yields further compression [2]. The first level partitions the permutation into *strict ascending runs* (maximal ranges of positions satisfying $\pi(i+k) = \pi(i)+k$). The second level partitions the *heads* (first position) of those strict runs into conventional ascending runs. This is analogous to the notion of Blocks described by Moffat and Petersson [7] for multisets.

Theorem 5. *For any strictly positive integer n and any permutation π on $[1..n]$ which can be decomposed into* $\texttt{nBlock}$ *strict runs and into* $\texttt{nRuns} \leq \texttt{nBlock}$ *monotone runs, let* $\texttt{vHRuns}$ *be the vector formed by the* $\texttt{nRuns}$ *monotone run lengths in the permutation of strict run heads. Then, there is a representation of π using at most* $\texttt{nBlock}\mathcal{H}(\texttt{vHRuns})) + O(\texttt{nBlock} \log \frac{n}{\texttt{nBlock}}) + o(n)$ *bits to support the operator $\pi()$ and $\pi^{-1}()$ in time within $O(1+\log \texttt{nBlock})$. This compressed data structure can be computed in time within $O(n(1+\log \texttt{nBlock}))$.*

Shuffled Subsequences The preorder measures seen so far have considered runs which group contiguous positions in π: this does not need to be always the case. A permutation π over $[1..n]$ can be decomposed in n comparisons into a minimal number $\texttt{nSUS}$ of *Shuffled Up Sequences*, defined as 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 [8]. Then those subsequences can be merged using the same techniques as above, which yields a new adaptive sorting algorithm and a new compressed data structure [2]. An optimal partition of a permutation π over $[1..n]$ into a minimal number $\texttt{nSMS}$ of *Shuffled Monotone Sequences*, sequences of not necessarily consecutive subsequences of increasing or decreasing numbers, is NP-hard to compute [9] but if such a permutation is given, the same technique applies [10].

LRM Subsequences LRM-Trees partition a sequence of values into consecutive sorted blocks, and express the relative position of the first element of each block within a previous block. Such a tree can be computed in $2(n-1)$ comparisons within the array and overall linear time, through an algorithm similar to that of Cartesian Trees [11]. The interest of LRM trees in the context of adaptive sorting and permutation compression is that the values are increasing in each root-to-leaf branch: they form a partition of the array into sub-sequences of increasing values. Barbay et al. [3] described how to compute the partition of the LRM-tree of minimal size-vector entropy, which yields a compressed data structure asymptotically smaller than $\mathcal{H}(\texttt{vRuns})$-adaptive sorting, and smaller in practice than $\mathcal{H}(\texttt{vSUS})$-adaptive sorting; as well as a faster adaptive sorting algorithm.

Number of Inversions The preorder measure $\texttt{nInv}$ counts the number of pairs (i,j) of positions $1 \leq i < j \leq n$ in a permutation π over $[1..n]$ such that $\pi(i) > \pi(j)$. Its value is exactly the number of comparisons performed by the algorithm `Insertion Sort`, between n and n^2 for a permutation over $[1..n]$. A variant of `Insertion Sort`, named `Local Insertion Sort`, sorts π in $n(1+\lceil \lg(\texttt{nInv}/n) \rceil)$ comparisons [7, 6].

Simply encoding the n values $(\pi(i)-i)_{i \in [1..n]}$ using the γ' code from Elias [12], and indexing the positions of the beginning of each code by a compressed bit vector yields a compressed data structure supporting the operator $\pi()$ in constant time. The resulting data structure uses space within $n(1+2\lg \frac{\texttt{nInv}}{n}) + o(n)$ bits. Support for the operator $\pi^{-1}()$ can be added in two distinct ways, either encoding both π and π^{-1} using this technique within $2n(1+2\lg \frac{\texttt{nInv}}{n}) + o(n)$ bits, which supports both operators $\pi()$ and $\pi^{-1}()$ in constant time; or adding support for the operator $\pi^{-1}()$ using Munro et al.'s shortcut succinct index for permutations [1] described previously.

Removing Elements The preorder measure nRem counts the minimum number of elements that must be removed from a permutation so that what remains is already sorted. Its exact value is n minus the length of the *Longest Increasing Subsequence*, which can be computed in time within $O(n \log n)$. Alternatively, the value of nRem can be approximated within a constant factor of 2 in $2(n-1)$ comparisons. Partitioning π into the removed elements and the remaining ones through a bit vector of n bits; representing the order of the 2nRem elements in a wavelet tree (using any of the data structures described above); and representing the merging of both into n bits; yields a compressed data structure using space within $2n + 2\mathtt{nRem}\lg(n/\mathtt{nRem}) + o(n)$ bits and supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time, within $O(1+\log(\mathtt{nRem}+1))$.

Applications

Integer Functions

Munro et al. [1] extended the results on permutations to arbitrary functions from $[1..n]$ to $[1..n]$. Again $f^k(i)$ indicates the function iterated k times starting at i: if k is non-negative, this is straightforward. The case in which k is negative is more complicated as the image is a (possibly empty) multiset over $[1..n]$.

Whereas π is a set of cycles, f can be viewed as a set of cycles in which each node is the root of a tree. Starting at any node (element of $[1..n]$), the evaluation moves one step along a branch of the tree, or one step along a cycle. Moving k steps in a positive direction is straightforward, one moves up a tree and perhaps around a cycle. When k is negative, one must determine all nodes at distance k from the starting location, i, in the direction towards the leaves of the trees. The key technical issue is to run across succinct tree representations picking off all nodes at the appropriate levels. Using a raw encoding of the permutation mapping integers to the nodes, and Munro et al.'s shortcut succinct index [1] to support the operations on it yields the following result:

Theorem 6. *For any fixed ε, $n > 0$ and $f : [1..n] \to [1..n]$ there is a representation of f using $(1+\varepsilon)n \lg n + O(1)$ bits of space to computes $f^k(i)$ in time within $O(1+|f^k(i)|)$, for any integer k and for any integer $i \in [1..n]$.*

Open Problems

Other measures of disorder

Moffat and Petersson [7] list many measures of preorder and adaptive sorting techniques. Each measure explored above yields a compressed data structure for permutations supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time. 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 the inverse permutation. More work is required in order to decide whether there are compressed data structures for permutations, supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time and using space proportional to the other preorder measures[7, 6] (e.g. Reg, Exc, Block and Enc).

Sorting and Encoding Multisets

Munro and Spira [13] showed how to sort multisets through `MergeSort`, `Insertion Sort` and `Heap Sort`, adapting them with counters to sort in time within $O(n(1 + \mathcal{H}(\langle m_1, \ldots, m_r \rangle)))$ where m_i is the number of occurrences of i in the multiset (note that this is orthogonal to the results described in this chapter, that depend on the distribution of the lengths of monotone runs). It seems easy to combine both approaches (e.g. on `MergeSort` in a single algorithm using both runs and counters), yet quite hard to *analyze* the complexity of the resulting algorithm and compressed data structure. The difficulty measure must depend not only on both the entropy of the partition into runs and the entropy of the partition of the values of the elements, but also on the interaction of those partitions.

Compressed Data Structures Supporting $\pi^k()$

In Munro et al.'s "Cycle" data structure [1] for supporting the operator $\pi^k()$ (Theorem 2), the raw encoding of the permutation σ representing the cycles of π can be replaced by any compressed data structure such as those described here, with the warning that the compressibility of σ depends not only on π but also on the order in which its cycles are placed in σ. The question to know if there is a compressed data structure supporting the operator $\pi^k()$ which takes advantage of this order is open.

Recommended Reading

1. J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012.
2. Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science (TCS)*, 513:109–123, 2013.
3. Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *ELSEVIER Theoretical Computer Science (TCS)*, 459:26–41, 2012.
4. Jérémy Barbay. From time to space: Fast algorithms that yield small and fast data structures. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms (IanFest)*, volume 8066 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2013.
5. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.
6. Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.
7. Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24(2):70–77, 1992.
8. Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. In *Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 181–191, London, UK, 1990. Springer-Verlag.
9. Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994.
10. Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.
11. Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM Press, 1984.
12. P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.
13. J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM Journal of Computation*, 5(1):1–8, 1976.