Succinct Encoding of Permutations and its Applications to Text Indexing (2003; Munro, Raman, Raman, Rao)

Jérémy Barbay (University of Waterloo), www.cs.uwaterloo.ca/~jbarbay
J. Ian Munro (University of Waterloo), www.cs.uwaterloo.ca/~imunro
entry editor: Paolo Ferragina

INDEX TERMS: Succinct Data Structures, Permutations, Functions, Text Strings, Binary Relations
SYNONYMS: None is reported.

1 PROBLEM DEFINITION

A succinct data structure for a given data type is a representation of the underlying combinatorial object that uses an amount of space "close" to the information theoretic lower bound, together with algorithms that supports operations of the data type "quickly". A natural example is the representation of a binary tree [5]: an arbitrary binary tree on n nodes can be represented in $2n + o(n)$ bits while supporting a variety of operations on any node, which include finding its parent, its left or right child, and returning the size of its subtree, each in time within $O(1)$. As there are $\binom{2n}{n}/(n+1)$ binary trees on n nodes and the logarithm of this term[1] is $2n - o(n)$, the space used by this representation is optimal to within a lower order term.

In the applications considered in this entry, the principle concern is with indexes supporting search in strings and in XML-like documents (i.e. tree-structured objects with labels and "free text" at various nodes). As happens not only labeled trees but also arbitrary binary relations over finite domains are key building blocks for this. Preprocessing such data-structures to efficiently support searches is a complex process, requiring a variety of subordinate structures.

A basic building block for this work 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 $\lg(n!)$. The interesting problem is to support both the permutation $\pi()$ and its inverse $\pi^{-1}()$: namely, how to represent an arbitrary permutation π on $[1..n]$ in a succinct manner so that the operator $\pi^k(i)()$ ($\pi()$ iteratively applied k times starting at i, where k can be any integer and $\pi^{-1}()$ is the inverse of $\pi()$) can be evaluated quickly.

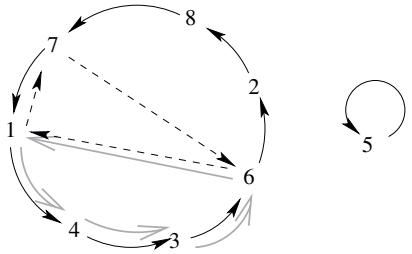
2 KEY RESULTS

Munro *et al.* [7] studied the problem of succinctly representing a permutation to support the operator $\pi^k()$ quickly. They give two solutions: one supports the operator arbitrarily quickly, at the cost of extra space; the other uses essentially optimal space at the cost of slower support.

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 shortcuts

[1] All logarithms are taken to the base 2. By convention, the iterated logarithm is denoted by $\lg^{(i)} n$, hence $\lg \lg \lg x$ is $\lg^{(3)} x$.

or back pointers. In each cycle of length at least t, every t-th element has a pointer t steps back. $\pi(i)$ 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 $o(n)$ 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 operations on binary strings [8]). This gives the location of the appropriate back pointer in the auxiliary array S.



i		1	2	3	4	5	6	7	8
$\pi(i)$	=	4	8	6	3	5	2	1	7
B	=	1	0	0	0	0	1	1	0
S	=	7					1	6	

Figure 1: A permutation on [1..8], with 2 cycles and 3 back pointers. The full lines correspond to the permutation, the dashed lines to the back pointers, the grey lines to the edges traversed to compute $\pi^{-1}(3)$.

For example, the permutation $\pi = (4, 8, 6, 3, 5, 2, 1, 7)$ on [1..8] corresponds to a set of two cycles, $\{(\mathbf{1}, 4, 3, \mathbf{6}, 2, 8, \mathbf{7}), (5)\}$ (see Figure 1 for a graphical representation of π and of those cycles).

For $t = 3$, the back pointers are cycling backward between 1, 6 and 7 in the largest cycle (there are none in the other because it is smaller than t). To find $\pi^{-1}(3)$, follow π from 3 to 6, observe that 6 is a back pointer because marked by the second 1 in B, and follow the second value of S to 1, then follow π from 1 to 4 and then to 3: the predecessor of 3 has been found. As there are back-pointer every t elements in the cycle, finding the predecessor requires $O(t)$ memory accesses.

For arbitrary i and k, $\pi^k(i)$ 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. For example, the permutation $\pi = (6, 4, 3, 5, 2, 1)$ on [1..6] has three cycles $\{(1, 6), (3), (2, 5, 4)\}$ and is encoded by the permutation $\sigma = (1, 6, 3, 2, 5, 4)$ on the same domain [1..6] and the bit vector $B = (1, 0, 1, 1, 0, 0)$. 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 operations on B. Then $\pi^k(i)$ is determined by taking k modulo the cycle length and moving that number of steps around the cycle starting at the position of i.

Other than the support of the inverse of σ, all operations are performed in constant time, hence the total time depends on the value chosen for t.

Theorem 1 (Munro et al. 2003)**.** *There is a representation of an arbitrary permutation π on $[1..n]$ using at most $(1+\varepsilon)n \lg n + O(n)$ bits that can support the operation $\pi^k()$ 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. $O(n/\lg n)$ extra words), time within $\Omega(\lg n)$ is necessary to support both $\pi()$ and $\pi^{-1}()$. In the RAM model, Munro *et al.* [7] described a data structure occupying even less space using Benes Network, a communication network composed of switches that can be used to implement permutations.

Theorem 2 (Munro *et al.* 2003)**.** *There is a representation of an arbitrary permutation π on $[1..n]$ using at most $\lceil \lg(n!) \rceil + O(n)$ bits that can support the operation $\pi^k()$ in time within $O(\lg n / \lg^{(2)} n)$.*

While this data-structure uses less space than the other, it requires more time for each operation. It is not known whether this time bound can be improved using only $O(n)$ "extra space". As a consequence, the first data structure is used in all applications. Obviously, any other solution can be used, potentially with a better time/space trade-off.

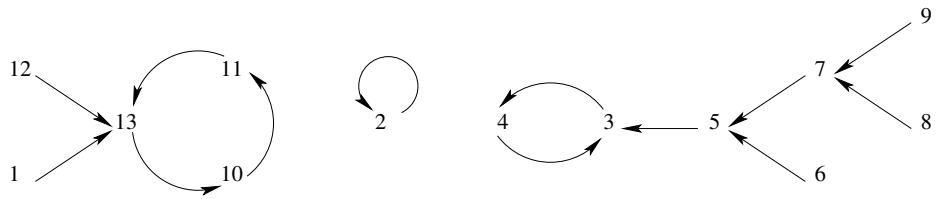
3 APPLICATIONS

Applications of those results on permutations can be grouped in several families: those concerning the encoding and indexing of integer functions (Section 3.1); those concerning the encoding and indexing of text strings (Section 3.2); and those concerning the encoding sparse binary relations and labeled trees (Section 3.3).

3.1 Functions

Munro and Rao [9] 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 nonnegative, this is straightforward. The case in which k is negative is more interesting as the image is a (possibly empty) multiset over $[1..n]$ (see Figure 2 for an example). 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 toward the root of the tree or one step along a cycle (e.g. $f(8) = 7, f(10) = 11$ in Figure 2). Moving k steps in a positive direction is straightforward, one moves up a tree and perhaps around a cycle (e.g. $f^5(9) = f^2(3) = 3$). When k is negative one must determine all nodes of distance k from the starting location, i, in the direction towards the leaves of the trees (e.g. $f^{-1}(13) = \{1, 11, 12\}$, $f^{-1}(3) = \{4, 5\}$). The key technical issue is to run across succinct tree representations picking off all nodes at the appropriate levels.

Theorem 3 (Munro and Rao 2004)**.** *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 $i \in [1..n]$.*



i		1	2	3	4	5	6	7	8	9	10	11	12	13
$f(i)$	=	13	2	4	3	3	5	5	7	7	11	13	13	10
$f^{-1}(i)$	=	{}	{2}	{4,5}	{3}	{6,7}	{}	{8,9}	{}	{}	{13}	{10}	{}	{1,11,12}

Figure 2: A function on [1..13], with 3 cycles and 2 nontrivial tree structures.

3.2 Text Strings

Indexing text strings to support the search for patterns is an important general issue. Barbay *et al.* [2] consider "negative" searches, along the following lines:

Definition 1. *Consider a string $S[1..n]$ over the alphabet $[1..l]$. A position $x \in [1..n]$* matches a literal *$\alpha \in [1..l]$ if $S[x] = \alpha$. A position $x \in [1..n]$* matches a literal *$\overline{\alpha}$ if $S[x] \neq \alpha$. The set $\{\overline{1}, \ldots, \overline{l}\}$ is denoted by $[\overline{1..l}]$.*

Given a string S of length n over an alphabet of size l, for any position $x \in [1..n]$ in the string, any literal $\alpha \in [1..l] \cup [\overline{1..l}]$ and any integer r, consider the following operators:

- $\texttt{string\_rank}_\texttt{S}(\alpha, \texttt{x})$: returns the number of occurrences of α in $S[1..x]$;
- $\texttt{string\_select}_\texttt{S}(\alpha, \texttt{r})$: returns the position of the r-th occurrence of α in S, or ∞ if none exists;
- $\texttt{string\_access}_\texttt{S}(\texttt{x})$: returns the label $S[x]$;
- $\texttt{string\_pred}_\texttt{S}(\alpha, \texttt{x})$: returns the last occurrence of α in $S[1..x]$, or ∞ if none exists; and
- $\texttt{string\_succ}_\texttt{S}(\alpha, \texttt{r})$: returns the first occurrence of α in $S[x..n]$, or ∞ if none exists.

Golynski *et al.* [4] observed that a string of length l on alphabet $[1..l]$ can be encoded and indexed by a permutation on $[1..l]$ (which for each label lists the positions of all its occurrences) together with a bit vector of length $2l$ (which encode in unary the numbers of occurences of each symbol, in alphabetical order). For instance, the string $ACCA$ on alphabet $\{A, B, C, D\}$ is encoded by the permutation $(1, 4, 2, 3)$ and the bit vector $(0, 0, 1, 1, 0, 0, 1, 1)$. Such encoding supports the operators rank, select and access in time within $O(\lg^{(2)} n)$, by using a value of $t = \lg^{(2)} n$ in the encoding of permutation of Theorem 1.

This encoding achieves fast support for variants of those search operators for labels, with a small overhead in space, by integrating the encodings of the text and the indexing information. Barbay *et al.* [2] extended those operators to literals, and showed how to separate the *succinct encoding* of the string S, in a manner that assumes we can access a word of S in a fixed time bound, and a *succinct index* containing auxiliary information useful to support the search operators defined above.

Theorem 4 (Barbay *et al.* 2007)**.** *Given access to a label in the raw encoding of a string $S \in [1..l]^n$ in time $f(n, l)$, there is a succinct index using $n(1 + o(\lg l))$ bits that supports the operators $\texttt{string\_rank}_\texttt{S}$, $\texttt{string\_pred}_\texttt{S}$ and $\texttt{string\_succ}_\texttt{S}$ for any literal $\alpha \in [1..l] \cup [\overline{1..l}]$ in time within $O(\lg^{(2)} l \cdot \lg^{(3)} l \cdot (f(n, l) + \lg^{(2)} l))$; and the operator $\texttt{string\_select}_\texttt{S}$ for any label $\alpha \in [1..l]$ in time within $O(\lg^{(3)} l \cdot (f(n, l) + \lg^{(2)} l))$.*

The separation between the encoding of the string or an XML-like document and its index has two main advantages:

- The string can now be compressed and searched at the same time, provided that the compressed encoding of the string supports the access in reasonable time, as does the one described by Ferragina and Venturini [3].
- The operators can be supported for several orderings of the string, for instance induced by distinct traversals of a labeled tree, with only a small cost in space. It is important for instance when those orders correspond to various traversals of a labeled structure, such as the depth-first and DFUDS traversals of a labeled tree [2].

3.3 Binary Relations

Given two ordered sets of sizes l and n, denoted by $[1..l]$ and $[1..n]$, a binary relation R between these sets is a subset of their Cartesian product, i.e. $R \subset [1..l] \times [1..n]$. It is used, for instance, to represent the relation between a set of labels $[1..l]$ and a set of objects $[1..n]$.

Although a string can be seen as a particular case of a binary relation, where the objects are positions and exactly one label is associated to each position, the search operations on binary relations are diverse, including operators on both the labels and the objects. For any literal α, object x, and integer r, consider the following operators:

- $\texttt{label\_rank}_{\texttt{R}}(\alpha, \texttt{x})$: returns the number of objects labeled α preceding or equal to x;
- $\texttt{label\_select}_{\texttt{R}}(\alpha, \texttt{r})$: returns the position of the r-th object labeled α if any, or ∞ otherwise;
- $\texttt{label\_nb}_{\texttt{R}}(\alpha)$: returns the number of objects with label α;
- $\texttt{object\_rank}_{\texttt{R}}(\texttt{x}, \alpha)$: returns the number of labels associated with object x preceding or equal to label α;
- $\texttt{object\_select}_{\texttt{R}}(\texttt{x}, \texttt{r})$: returns the r-th label associated with object x, if any, or ∞ otherwise;
- $\texttt{object\_nb}_{\texttt{R}}(\texttt{x})$: returns the number of labels associated with object x; and
- $\texttt{table\_access}_{\texttt{R}}(\texttt{x}, \alpha)$: returns checks whether object x is associated with label α.

Barbay *et al.* [1] observed that such a binary relation, consisting of t pairs from $[1..n]\times[1..l]$, can be encoded as a text string S listing the t labels, and a binary string B indicating how many labels are associated with each object. So search operations on the objects associated with a fixed label are reduced to a combination of operators on text and binary strings. Using a more direct reduction to the encoding of permutations, the index of the binary relation can be separated from its encoding, and even more operators can be supported [2]:

Theorem 5 (Barbay *et al.* 2007)**.** *Given support for* $\texttt{object\_access}_{\texttt{R}}$ *in time* $f(n, l, t)$ *on a binary relation formed by* t *pairs from an object set* $[1..n]$ *and a label set* $[1..l]$*, there is a succinct index using* $t(1+o(\lg l))$ *bits that supports* $\texttt{label\_rank}_{\texttt{R}}$ *for any literal* $\alpha \in [1..l]\cup\overline{[1..l]}$ *and* $\texttt{label\_access}_{\texttt{R}}$ *for any label* $\alpha \in [1..l]$ *in time within* $O(\lg^{(2)} l \cdot \lg^{(3)} l \cdot (f(n, l, t) + \lg^{(2)} l))$*, and* $\texttt{label\_select}_{\texttt{R}}$ *for any label* $\alpha \in [1..l]$ *in time within* $O(\lg^{(3)} l \cdot (f(n, l, t) + \lg^{(2)} l))$.

To conclude this entry, note that a labeled tree T can be represented by an ordinal tree coding its structure [6] and a string S listing the labels of the nodes. If the labels are listed in preorder (resp. in DFUDS order) the operator $\texttt{string\_succ}_{\texttt{S}}$ enumerates all the descendants (resp. children) of a node matching some literal α. Using succinct indexes, a single encoding of the labels and the support of a permutation between orders is sufficient to implement both enumerations, and other search operators on the labels. These issues, along with strings and labeled trees compression techniques which achieve the entropy of the indexed data, are covered in more details in other entries cited in the Cross References section.

4 OPEN PROBLEMS

None is reported.

5 EXPERIMENTAL RESULTS

None is reported.

6 DATA SETS

None is reported.

7 URL to CODE

None is reported.

8 CROSS REFERENCES

Compressed Text Indexing; Compressed Suffix Array; Rank and Select over Binary Strings; Text Indexing, Tree Compression and Indexing.

9 RECOMMENDED READING

[1] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao, *Adaptive searching in succinctly encoded binary relations and tree-structured documents*, in Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM), vol. 4009 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 2006, pp. 24–35.

[2] J. Barbay, M. He, J. I. Munro, and S. S. Rao, *Succinct indexes for strings, binary relations and multi-labeled trees*, in Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, 2007, pp. 680–689.

[3] P. Ferragina and R. Venturini, *A simple storage scheme for strings achieving entropy bounds*, in Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, 2007, pp. 690–695.

[4] A. Golynski, J. I. Munro, and S. S. Rao, *Rank/select operations on large alphabets: a tool for text indexing*, in Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, 2006, pp. 368–373.

[5] G. Jacobson, *Space-efficient static trees and graphs*, in Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.

[6] J. Jansson, K. Sadakane, and W.-K. Sung, *Ultra-succinct representation of ordered trees*, in Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, 2007, pp. 575–584.

[7] J. I. Munro, R. Raman, V. Raman, and S. S. Rao, *Succinct representations of permutations*, in Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP), vol. 2719 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 2003, pp. 345–356.

[8] J. I. Munro and V. Raman, *Succinct representation of balanced parentheses and static trees*, SIAM Journal on Computing, 31 (2001), pp. 762–776.

[9] J. I. Munro and S. S. Rao, *Succinct representations of functions*, in Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), vol. 3142 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 2004, pp. 1006–1015.