Path Queries on Functions [1]

Travis Gagie [2,3]

Center for Biotechnology and Bioengineering (CeBiB)
School of Computer Science and Telecommunications, Diego Portales University, Chile
travis.gagie@gmail.com

Meng He

Faculty of Computer Science, Dalhousie University, Canada
mhe@cs.dal.ca

Gonzalo Navarro [2,4]

Center for Biotechnology and Bioengineering (CeBiB)
Millenium Institute for Foundational Research on Data
Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract

Let $f : [1..n] \to [1..n]$ be a function, and $\ell : [1..n] \to [1..\sigma]$ indicate a label assigned to each element of the domain. We design several compact data structures that answer various kinds of summary queries on the labels of *paths* in f. For example, we can find either the minimum label in $f^k(i)$ for a given i and any $k \geq 0$ in a given range $[k_1..k_2]$, or the minimum label in $f^{-k}(i)$ for a given i and $k > 0$, using $n \lg n + n \lg \sigma + o(n \lg n)$ bits and time $O(\alpha(n))$, the inverse Ackermann function. Within similar space we can count, in time $O(\lg n / \lg \lg n)$, the number of labels within a range, and report each element with such labels in $O(\lg n / \lg \lg n)$ additional time. Several other tradeoffs and possible queries are considered, such as selection, top-r queries and τ-majorities. Finally, we consider queries that allow us navigate on the graph of the function, such as the nearest common successor of two elements, or the nearest successor or predecessor of an element within a range of labels.

Key words: Succinct data structures; Integer functions; Range queries; Trees and permutations

1 Introduction

We focus on the representation of *integer functions* where the domain coincides with the image, $f : [1..n] \to [1..n]$. This kind of functions were studied by Munro et al. [21], who focused on how to compute efficiently *powers* of functions. A positive power is $f^k(i)$, for a given $i \in [1..n]$ and $k \geq 0$, whereas a negative power returns all the elements in the set $f^{-k}(i) = \{j, f^k(j) = i\}$, for a given $i \in [1..n]$ and $k > 0$. They show that f can be represented within $n \lg n(1+1/t)+O(n)$ bits so that any positive power $f^k(i)$ is computed in time $O(t)$, and any negative power $f^{-k}(i)$ is listed in time $O(t + |f^{-k}(i)|)$, for any $t = O(\lg n)$. The main idea of Munro et al. is summarized in their metaphor "functions are just hairy permutations", in the sense that the directed graph $G(V, E)$ where $V = [1..n]$ and $E = \{(i, f(i)), i \in [1..n]\}$ has the form of a set of cycles, where a tree may sprout from each node in each cycle (permutations, instead, are decomposed into just a set of cycles).

In this article we go beyond the goal of simply listing the elements of powers of permutations. Instead, we seek to compute *summaries* on the elements belonging to *paths* in G. We consider three kinds of paths P:

(1) A *positive path* is formed by the distinct elements in $f^{k_1..k_2}(i) = \{f^k(i), k \in [k_1..k_2]\}$ for a given $i \in [1..n]$ and $0 \leq k_1 \leq k_2$.
(2) A *negative path* is formed by the distinct elements in $f^{-k_1..-k_2}(i) = \{j \in f^{-k}(i), k \in [k_1..k_2]\}$ for a given $i \in [1..n]$ and $0 < k_1 \leq k_2$.
(3) A *negative path point* is a particular case of a negative path, formed by the elements in $f^{-k}(i)$, for a given $i \in [1..n]$ and $k > 0$.

In turn, we consider various kinds of summarizations. For maximum generality, let us assume that the domain elements are assigned a label $\ell : [1..n] \to [1..\sigma]$, and we perform summary queries on the labels. A particular case is that of $\ell(i) = i$. We consider the following queries on paths P:

(1) Minimum or maximum queries: Return $\min\{\ell(j), j \in P\}$ or $\max\{\ell(j), j \in P\}$.
(2) Selection queries: Return an element of P with the rth smallest or largest label; this includes queries where the value of r is relative to $|P|$, such as the median.
(3) Top-r queries: Return a set $M \subseteq P$ formed by r elements with largest labels in P, or r elements with smallest labels in P.
(4) τ-Majority queries: Return a set of labels whose relative frequency in P

[1] An early partial version of this paper appeared in *Proc. CPM 2017* [11].
[2] Partially funded by Basal Funds FB0001, Chile.
[3] Partially funded by Fondecyt Grant 1-171058, Chile.
[4] Partially funded by Millenium Institute for Foundational Research on Data, Chile.

is over τ, for a given $0 \leq \tau < 1$.

(5) Range queries: Let $R = \{j \in P, \ell(j) \in [\ell_1, \ell_2]\}$, given $1 \leq \ell_1 \leq \ell_2 \leq \sigma$. A counting query asks for $|R|$, whereas a reporting query requires listing all the elements in R.

As an application of summary queries on paths, suppose we are simulating a system to prepare for situations in which we need to react quickly, say, natural disasters or conflicts or critical-equipment failures. We run our simulation through some finite set of states and want to store the traces such that later, given a start state in that set and a number of time-steps, we can quickly return summaries about the states the simulation passes through from that state in that many steps. Of course, we could precompute all the possible answers, but this could take space quadratic in the number of states; we could iterate through all the relevant states at query time, but this could take linear time. If our simulation is deterministic, our problem reduces to storing a function (from states to states, with each state labelled by satellite data) compactly such that we can efficiently answer path queries on it.

The case of positive paths is the most direct. We build on recent results [16,5] that give succinct (and also larger) structures for various path queries on trees. Then a relatively simple unrolling and doubling of the cycles in the graph G allows us to directly apply their results to positive paths, with a small extra time penalty to map from the domain of f to the nodes of G. For example, using $n \lg n + n \lg \sigma + o(n \lg n)$ bits we can solve minimum or maximum queries in time $O(\alpha(n))$ (the inverse Ackermann function), range queries in time $O(\lg n / \lg \lg n)$ per returned datum and, raising the space to $n \lg n + 2n \lg \sigma + o(n \lg n)$ bits, selection queries in time $O(\lg n / \lg \lg n)$.

For negative path points, we unroll the cycles in such a way that all the desired nodes in any $f^{-k}(i)$ belong to a contiguous range within a single level of the tree. Then an appropriate layout of the data associated with the node allows us to reduce queries on negative path points to array range queries, and therefore reuse all the known results on array range queries [4,10,14,2], some of which do not even require us to represent the labels. Therefore, we generally obtain better results on negative path points than on positive paths. For example, we can solve minimum or maximum queries in time $O(\alpha(n))$ using only $n \lg n + o(n \lg n)$ bits. We can also solve top-r queries using $n \lg n + O(n \lg R)$ bits, where R is the maximum r value permitted, in time $O(r \lg_R n)$. As another example, we can solve τ-majority queries using $n \lg n + (1 + \epsilon) n \lg \sigma$ bits, for any constant $\epsilon > 0$, in time $O(1/\tau + \lg_\sigma n)$.

The hardest case are the general negative paths. Our queries in this case are mapped into a couple of two-dimensional spaces with labels, and thus the structures require from $O(n \lg n)$ to $O(n \lg^2 n)$ bits in order to offer polylogarithmic query times [19,6,24,23]. For example, we can solve top-r queries using

$O(n \lg n)$ bits and $O((r + \lg n) \lg^\epsilon n)$ time, for any constant $\epsilon > 0$. For range counting and reporting, we need $O(n \lg^2 n / \lg \lg n)$ bits and $O((\lg n / \lg \lg n)^2)$ time per datum retrieved.

Finally, we consider new queries to navigate on the values of the function, more sophisticated than the basic iteration queries supported by Munro et al. [21]. For example, we can find the nearest element that can be reached from two given elements i and i' through applications of f, in time $O(\lg n / \lg \lg n)$, using $n \lg n + O(n)$ bits. Within $n \lg n + n \lg \sigma + o(n \lg n)$ bits, we can find the nearest element reachable from an element i, whose label is within a given range, in time $O(1 + \lg \sigma / \lg \lg n)$.

An early partial version of this paper appeared in *Proc. CPM 2017* [11]. This extended version includes pseudocode for all the nontrivial procedures, more applications of the generic results to specific queries, improved results on negative path queries, considerations for the special cases of queries without labels, and a new section on sophisticated navigation queries. In addition, we obtain in this paper some new results of independent interest on labeled tree ancestors (Theorem 7).

2 Background

2.1 Rank and select on bitvectors

A bitvector $B[1..n]$ can be represented in $n + o(n)$ bits so as to perform operations *rank* and *select* in constant time [18,7,20]. Operation $rank_b(B, i)$, for $b \in \{0, 1\}$ and $i \in [1..n]$, is the number of occurrences of bit b in $B[1..i]$. Operation $select_b(B, j)$, with $b \in \{0, 1\}$ and $j \in [1..rank_b(B, n)]$, is the position of the jth occurrence of bit b in B.

When B is sparse, having only $m \ll n$ 1s, a structure that computes only $rank_1(B, i)$ whenever $B[i] = 1$ (and is unable to access B or to compute *select*) will be of interest to us. It is called a monotone minimum perfect hash function [1] and can compute *rank* in time $O(1)$ using $O(m \lg \lg n)$ bits, or in time $O(\lg \lg n)$ using $O(m \lg \lg \lg n)$ bits.

2.2 Permutations and functions

Munro et al. [21] regard a permutation π on $[1..n]$ as a directed graph $G = (V, E)$, where $V = [1..n]$ and $E = \{(i, \pi(i)),\ i \in [1..n]\}$. This graph turns out to be a set of simple cycles, which correspond to the cycle decomposition

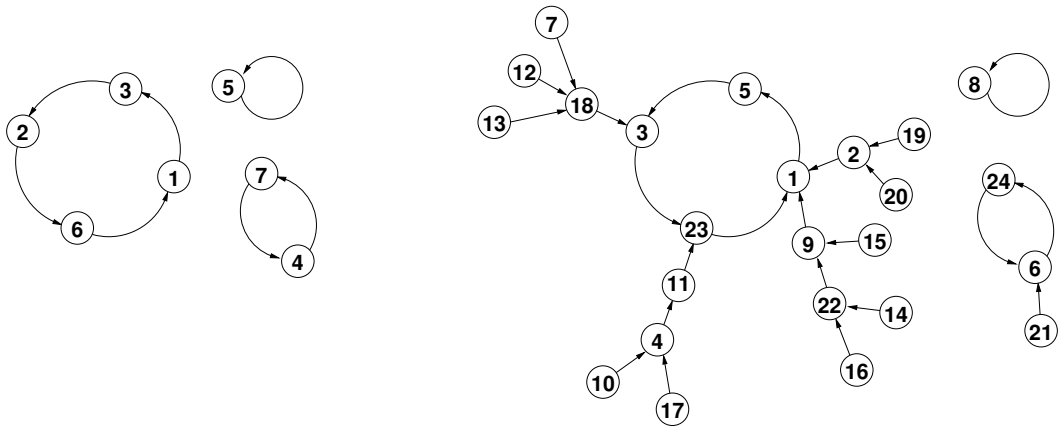

Fig. 1. On the left, the representation of a permutation as a directed graph. On the right, the permutation is extended into a function.

of π. Figure 1 (left) shows the grahical representation of permutation $\pi = (3\ 6\ 2\ 7\ 5\ 1\ 4)$, which is decomposed into the cycles $(1\ 3\ 2\ 6)$, (5), and $(4\ 7)$.

A function $f : [1..n] \to [1..n]$ is then regarded as an extension of permutations, where a general tree may sprout from each node of the cycles (with edges pointing towards the root). Figure 1 (right) illustrates the case of $f(1..24) = (5, 1, 23, 11, 3, 24, 18, 8, 1, 4, 23, 18, 18, 22, 9, 22, 4, 3,\ 2, 2, 6, 9, 1, 6)$, which extends the cycles of our example π.

From the results that are interesting to us, Munro et al. obtain two representations for permutations π. The first uses $\lg n! + o(n)$ bits and computes any $\pi(i)$ in time $O(1)$ and any $\pi^{-1}(i)$ in time $O(\lg n / \lg\lg n)$. The second uses $\lg n! + O((n/t)\lg n)$ bits, for any $t \leq \lg n$, and computes any $\pi(i)$ in time $O(1)$ and any $\pi^{-1}(i)$ in time $O(t)$. For functions, they can compute any positive power $f^k(i)$, with $k \geq 0$, or negative power $f^{-k}(i) = \{j,\ f^k(j) = i\}$, with $k > 0$, in time $O(t)$ and $O(t+|f^{-k}(i)|)$, respectively, using $n \lg n(1+1/t)+O(n)$ bits of space, for any $t = O(\lg n)$.

2.3 Path queries on trees

He et al. [16] and Chan et al. [5] recently showed how to represent a tree where the nodes have labels (or weights) in succinct space so as to support various queries on the paths of the tree. Let us regard the trees as acyclic connected graphs $G(V, E)$; then a *path* is a sequence of nodes $v_1, v_2, \ldots, v_p$, such that every $(v_k, v_{k+1}) \in E$, and it can be specified by giving v_1 and v_p.[5] Given a

[5] They actually handle undirected graphs, supporting paths between any two nodes u and v. Those can be easily decomposed into two directed paths, from u to $lca(u, v)$ and from v to $lca(u, v)$, where *lca* is the lowest common ancestor operation on the tree induced by choosing an arbitrary root from the graph nodes.

general ordinal tree of n nodes, where each node v has a label $\ell(v) \in [1..\sigma]$, they support the following queries on paths P of the tree that are of interest to us:

(1) Minimum/maximum queries, that is, finding a node with the smallest or largest label in P, are solved in time $\alpha(m,n)$ with a structure using $O(m)$ bits of space on top of the raw data, for any $m \geq n$, where α is the inverse of the Ackermann function [5]. By "raw data" we mean that the label of any node must be obtained in $O(1)$ time from its preorder.
(2) Selection, that is, find the node holding the rth smallest label in P, is solved in time $O(\lg\sigma/\lg\lg\sigma)$, with a structure using $nH(\ell) + o(n\lg\sigma) + O(n)$ bits of space. Here $H(\ell) \leq \lg\sigma$ is the entropy of the distribution of the values $\ell(v)$ over all the nodes v [16].
(3) Range queries include counting, that is, how many nodes in P have labels in $[\ell_1..\ell_2]$, and reporting, that is, listing all those nodes, given ℓ_1 and ℓ_2. Both are solved within $nH(\ell) + o(n\lg\sigma) + O(n)$ bits of space, supporting counting in time $O(1 + \lg\sigma/\lg\lg n)$ and reporting of r results in time $O((r+1)(1+\lg\sigma/\lg\lg n))$ [16]. By using more space, it is possible to match the same results of two-dimensional range queries [5].

Those structures include an $O(n)$-bit representation of the tree topology. There are several such representations, including balanced parentheses [18,22,25], DFUDS [3], LOUDS [18], and tree covering [12,15,8]. All of them use $2n+o(n)$ bits and support a wide set of navigation operations on trees. The representation used to solve these path queries [16,5] supports in constant time a few primitives that will be useful for our positive path queries:

- Mapping from each tree node v to a unique identifier $id(v) \in [1..n]$, and from an identifier $i \in [1..n]$ to the tree node, $node(i)$.
- Level ancestor queries, that is, given a node v and a distance d, $anc(v,d)$ is the ancestor of v at distance d (e.g., $anc(v,0)$ is v and $anc(v,1)$ is the parent of v).
- The depth of a node, $depth(v)$, where the depth of the root is 0.
- The leftmost leaf of the subtree of a node, $leftmost(v)$.
- The lowest common ancestor of two nodes, $lca(u,v)$.

For negative paths, instead, we will use the Fully-Functional representation [25], which represents the tree using $2n$ parentheses: the tree is traversed in depth-first order, writing an opening parenthesis when we reach a node and a closing one when we leave it. A node is identified with the position v of its opening parenthesis (thus, using preorders as node identifiers, we have $id(v) = rank_(v)$ and $node(i) = select_(i)$ on the parentheses sequence). Within $2n + o(n)$ bits, the representation supports in constant time all of the above operations, plus $fwd(x,d)$ and $bwd(x,d)$, defined as follows. Let $excess(x) = rank_(x) - rank_)(x)$ be the number of opening minus closing paren-

theses up to position x in the sequence (thus $depth(v) = excess(v) - 1$). Operation $fwd(x, d)$ (resp. $bwd(x, d)$) finds the closest position $y > x$ (resp. $y < x$) where $excess(y) = excess(x) + d$. For example, if there is an opening (resp. closing) parenthesis at x, its corresponding closing (resp. opening) parenthesis is at $close(x) = fwd(x, -1)$ (resp. $open(x) = bwd(x, 0) + 1$). Therefore, the number of nodes in the subtree of v is $subtreesize(v) = (close(v) - v + 1)/2$, and $anc(v, d) = bwd(v, -d - 1) + 1$. Also, the previous (resp. next) node at the same depth of x in preorder is $prev\text{-}level(v) = open(bwd(v, 0) + 1)$ (resp. $next\text{-}level(v) = fwd(close(v), 1)$). The operation $leftmost(v)$ can also be computed in this representation, with $select_)(rank_)(v) + 1) - 1$. Finally, the representation also computes in constant time the height of the subtree of a node, $height(v)$ (see the original article [25]).

2.4 Range queries on arrays

A much better studied particular case of path queries is that of range queries on an array $A[1..n]$ of labels in $[1..\sigma]$. The following is a brief selection from a number of results reported in the literature:

(1) Minimum queries, where it is possible to find the position of a minimum in any range $A[i..j]$ in $O(1)$ time with a structure that uses $2n + o(n)$ bits and does not access A [10]. An analogous result holds for maximum queries.
(2) Selection queries, where we can set at construction time a maximum value R of r that can be used in queries, and then a structure using $O(n \lg R)$ bits, without accessing A, can answer queries in optimal time $O(1 + \lg r / \lg \lg n)$ [14]. Note that we can set $R = n$ for maximum generality.
(3) Top-r queries, that is, finding r elements in $A[i..j]$ with the largest labels, can be answered in optimal time $O(r)$ with a structure that uses $O(n \lg R)$ bits and does not access A, where R is an upper bound on the values of r that can be queried [14].
(4) τ-Majority queries, that is, finding the labels whose relative frequencies in $A[i..j]$ are above τ. This can be solved in optimal time $O(1/\tau)$ and $O(1 + \epsilon)nH(\ell) + o(n)$ bits, for any constant $\epsilon > 0$; this representation contains A in compressed form. The space can be reduced to $nH(\ell)(1 + o(1)) + o(n)$ bits, and still obtain any time in $\omega(1/\tau)$ [2].
(5) Range counting can be performed in $O(1+\lg \sigma / \lg \lg n)$ time, and r results can be reported in time $O((r+1)(1+\lg \sigma / \lg \lg n))$, using $n \lg \sigma + o(n \lg \sigma)$ bits of space [4].

2.5 Range queries in two dimensions

When the ranges are two-dimensional and the points have weights, most of the queries require linear and even super-linear space. Some examples in the literature follow.

(1) The top-r heaviest elements in a two-dimensional range of an $n \times n$ grid with points having weights in $[1..\sigma]$ can be listed in time $O((r+\lg n)\lg^{\epsilon} n)$, for any constant $\epsilon > 0$, with a data structure that uses $O(n \lg n)$ bits [23, Lem. 7]. With $r = 1$, this gives a structure for range minima or maxima.
(2) The rth heaviest element in a two-dimensional range can be obtained in time $O(\ell \lg n \lg_{\ell} \sigma)$ with a structure using $n \lg n \lg_{\ell} \sigma + O(n \lg \sigma)$ bits, for any $\ell \in [2, \sigma]$ [24].
(3) The same structure of the previous point can be used to find the τ-majorities in a range in time $O((1/\tau)\ell \lg n \lg_{\ell} \sigma)$ [24].
(4) Range counting queries in three dimensions (or in two dimensions and labels) can be carried out in time $O((\lg n/\lg\lg n)^2)$ with a structure that uses $O(n \lg^2 n/\lg\lg n)$ bits of space [19]. Within that space, each point can be reported in time $O((\lg n/\lg\lg n)^2)$ [19]. By raising the space to $O(n \lg^{2+\epsilon} n)$ bits, for any constant $\epsilon > 0$, the time to report r points is reduced to $O(r + \lg\lg n)$ [6].

3 Positive Paths

A positive path of the form $f^{k_1..k_2}(i)$ can be handled by converting the graph G that represents f (recall Figure 1 (right)) into a single tree. The transformation is as follows:

(1) We cut each cycle $v_1 \to v_2 \to \ldots \to v_c \to v_1$ at an arbitrary position, say removing the edge $v_c \to v_1$. The result is a directed tree rooted at v_c (with arrows pointing from children to parents) where the cycle edges form the leftmost path.
(2) We add a new leaf per cycle, which will be the leftmost child of v_1.
(3) We add an artificial root, which will be the parent of the roots v_c of all the cycles.
(4) We represent the resulting tree using the data structures of Section 2.3 [16,5], for whichever query we want to answer. The representation supports in constant time the operations *id*, *node*, *anc*, *depth*, *leftmost*, and *lca*, which we need.
(5) We store a bitvector $B[1..n+l+1]$, where $l \leq n$ is the number of leaves added, or equivalently the number of cycles in f, so that $B[i] = 1$ iff the tree node with identifier i is one of the original nodes of G. We give

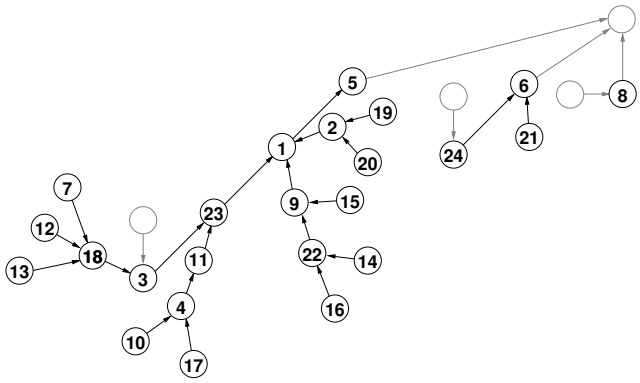

Fig. 2. Our transformation to solve positive queries on functions using path queries on trees.

rank and *select* support to B, so as to map the tree node identifiers in $[1..n+l+1]$ of the nodes that are in G to the interval $[1..n]$.

(6) We store a permutation π that goes from the mapped node identifiers in $[1..n]$ to the corresponding domain elements, using the representation of Section 2.2.

Figure 2 exemplifies our construction on the function of Figure 1. The permutation π is displayed in the form of numbers associated with the nodes. Note how we have broken the cycle $3 \to 23 \to 1 \to 5 \to 3$, for example.

Consider now a positive path query $f^{k_1..k_2}(i)$. In the simplest case, we proceed as follows:

(1) We compute $v = node(select_1(B, \pi^{-1}(i)))$, the node where the path query will start.
(2) We compute the path extremes $v_s = anc(v, k_1)$ and $v_e = anc(v, k_2)$.
(3) We carry out the desired query on the tree path from v_s to v_e.
(4) Any node u returned by the query is mapped back to a domain value in constant time using $\pi(rank_1(B, id(u)))$.

In our example, we can compute a query on $f^{1..3}(4) = (11, 23, 1)$ with this technique. However, consider $f^{1..4}(15)$. Our technique maps the path to the domain elements $(9, 1, 5, root)$, whereas the correct domain elements to include were $(9, 1, 5, 3)$. This is because the path goes through the node v_c where we have cut the cycle. In general, both k_1 and k_2 may be several times larger than the cycle length. The way we handle this situation is different for *decomposable* and *non-decomposable* queries: the former can be computed from the results obtained on disjoint subpaths, whereas the latter cannot. For example, range counting and reporting are decomposable, whereas range minimum is decomposable only if we store the labels, as in the solution of Section 2.3 [5] (but not in that of Section 2.4 [10]). Selection queries are not decomposable within the same query times obtained for a single path [5,14], even if we store the labels.

If the query we want to solve is decomposable, we use the cycle as follows. First, if $k_1 \geq depth(v)$, then we set $v_s \leftarrow anc(v_1, (k_1 - depth(v)) \bmod c)$, where v_1 is the lowest node of the cycle and c is the cycle length. Similarly, if $k_2 \geq depth(v)$, we set $v_e \leftarrow anc(v_1, (k_2 - depth(v)) \bmod c)$. For this we compute $v_c = anc(v, depth(v) - 1)$, then $v_1 = anc(leftmost(v_c), 1)$ and $c = depth(v_1)$.

However, v_e might not be an ancestor of v_s after this transformation, that is, $depth(v_e) > depth(v_s)$ or $anc(v_s, depth(v_s) - depth(v_e)) \neq v_e$. This means that the positive path is cut into two tree paths: one from v_s to v_c, and the other from v_1 to v_e. In our example, $f^{1..4}(15)$ is cut into the paths $(9, 1, 5)$ and (3).

A final issue is that, if $k_2 - k_1 \geq c$, we may visit the same domain values several times along the positive path. Since we want to consider each distinct element only once, we can solve this problem by splitting the query into up to three paths: one inside the tree where v belongs that sprouts from the cycle, and two on the cycle. We first compute $v' = lca(v, v_1)$, to find the cycle node where the tree of v sprouts. Then a first path to consider, if $k_1 < d = depth(v) - depth(v')$, is the one corresponding to $[k_1, \min(k_2, d-1)]$. If $k_2 \geq d$, then we consider paths on the cycle, starting at node v' and with the range $[k'_1, k'_2] = [\max(0, k_1 - d), k_2 - d]$. If $k'_2 - k'_1 \geq c - 1$, we simply include the whole cycle, with the path from v_1 to v_c. Otherwise, we do as before.

Algorithm 1 gives the complete procedure. It is possible to reduce the case of three paths to two, since those of lines 9 and 23 can be concatenated into one, but we opt for simplicity.

We have then the following theorem, where the extra time is the one spent to compute $\pi^{-1}(i)$ and the extra space is that of storing π and B.

Theorem 1. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a tree representation that performs in constant time the operations id, node, anc, depth, leftmost, and lca, and in addition it solves a certain decomposable path query on n-node trees with labels in $[1..\sigma]$ in $T(n, \sigma)$ time, using in total $S(n, \sigma)$ bits of space. Then, there exists a data structure using $n \lg n + O(n) + S(n, \sigma)$ bits that answers the same query on the positive paths of f in time $O(\lg n / \lg \lg n) + T(n, \sigma)$. There exists another data structure using $n \lg n(1 + 1/t) + O(n) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t > 0$.*

By considering the minimum/maximum queries of Chan et al. [5] and the range queries of He et al. [16] (Section 2.3), we obtain the following corollaries. For the first one, we use a compressed representation of the labels, in preorder, that accesses any value in $O(1)$ time [9].

Algorithm 1: Computing decomposable queries on positive paths.

Proc $Positive(i, k_1, k_2)$
  $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
  $v_c \leftarrow anc(v, depth(v) - 1)$
  $v_1 \leftarrow anc(leftmost(v_c), 1)$
  $c \leftarrow depth(v_1)$
  $v' \leftarrow lca(v_1, v)$
  $d \leftarrow depth(v) - depth(v')$
  if $k_1 < d$ **then**
    Compute path query from $anc(v, k_1)$ to $anc(v, \min(k_2, d-1))$
  if $k_2 \geq d$ **then**
    $k'_1 \leftarrow \max(0, k_1 - d)$
    $k'_2 \leftarrow k_2 - d$
    if $k'_2 - k'_1 \geq c - 1$ **then**
      Compute path query from v_1 to v_c
    else
      if $k'_1 < depth(v')$ **then** $v_s \leftarrow anc(v', k'_1)$
      else $v_s \leftarrow anc(v_1, (k'_1 - depth(v')) \bmod c)$
      if $k'_2 < depth(v')$ **then** $v_e \leftarrow anc(v', k'_2)$
      else $v_e \leftarrow anc(v_1, (k'_2 - depth(v')) \bmod c)$
      if $depth(v_s) \geq depth(v_e)$ **and** $anc(v_s, depth(v_s) - depth(v_e)) = v_e$ **then**
        Compute path query from v_s to v_e
      else
        Compute path query from v_s to v_c
        Compute path query from v_1 to v_e
  Return the composition of all the path queries performed; resulting nodes u are converted into domain values $\pi(rank_1(B, id(u)))$

Corollary 1. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + nH(\ell) + o(n \lg \sigma) + O(n)$ bits that answers minimum/maximum queries on the positive paths of f in time $O(\lg n / \lg \lg n)$, where $H(\ell) \leq \lg \sigma$ is the entropy of the distribution of the values in ℓ. There exists another structure using $n \lg n(1 + 1/t) + nH(\ell) + o(n \lg \sigma) + O(m)$ bits, for any $m \geq n$, that answers the queries in time $O(t + \alpha(m, n))$, for any $t > 0$. By setting $t = \alpha(m, n)$, we obtain a structure using $n \lg n + nH(\ell) + o(n \lg n) + O(m)$ bits that answers the queries in time $O(\alpha(m, n))$.*

Corollary 2. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + nH(\ell) + O(n) + o(n \lg \sigma)$ bits that answers counting*

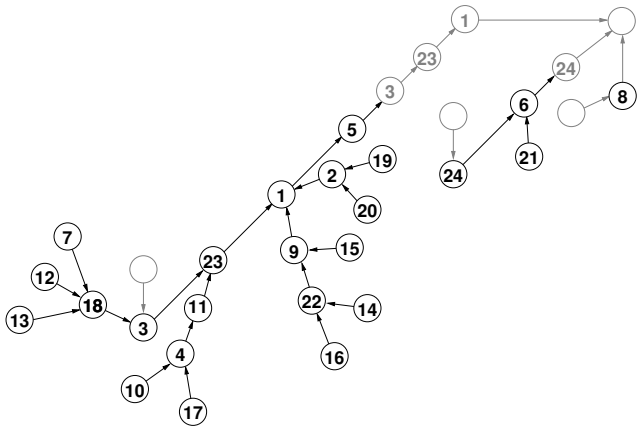

Fig. 3. Our transformation to solve non-decomposable positive queries on functions using path queries on trees.

queries on the positive paths of f in time $O(\lg n/\lg\lg n)$, and also reports those r results in time $O(\lg n/\lg\lg n + r(1+\lg\sigma/\lg\lg n))$, where $H(\ell) \leq \lg\sigma$ is the entropy of the distribution of the values in ℓ.

It is also possible to adapt the faster counting and reporting of Chan et al. [5], by using more space.

3.2 Non-decomposable path queries

When the query is not decomposable, we cannot allow splitting paths. Instead, we unroll the cycles twice, as illustrated in Figure 3. More formally:

(1) We cut each cycle $v_1 \to v_2 \to \ldots \to v_c \to v_1$ as before, removing the edge $v_c \to v_1$ and leaving a tree rooted at v_c.
(2) We add a leaf as the leftmost child of v_1, as before.
(3) We add an upward path per cycle, starting at each tree root v_c, which repeats the cycle with copies of the nodes. That is, we add edges $v_c \to v'_1 \to v'_2 \to \ldots \to v'_{c-1}$. Each of the new nodes v'_i is assigned the same label of v_i.
(4) We add an artificial root, which will be the parent of all the nodes v'_{c-1} (or of the node $v_1 = v_c$ for cycles of length 1, since in those cases no v'_i nodes are added).
(5) We represent the resulting tree using the data structures of Section 2.3, as before.
(6) We store a bitvector $B[1..n+g]$, where $g \leq n+1$ is the number of nodes added, so that $B[i] = 1$ iff the tree node with identifier i is one of the original nodes of G. As before, we give *rank* and *select* support to B.
(7) We store a permutation π that goes from the mapped node identifiers in $[1..n]$ to the corresponding domain elements, as before.

We can now compute $v'_{c-1} = anc(v, depth(v)-1)$, $v_1 = anc(leftmost(v'_{c-1}), 1)$, $c = (depth(v_1)+1)/2$, and $v_c = anc(v_1, c-1)$. We also compute $v' = lca(v_1, v)$ as before. There are two cases. The first is that the path starts inside the subtree sprouting from v', that is, $k_1 < d = depth(v) - depth(v')$. In this case, we set $v_s = anc(v, k_1)$. Then, if $k_2 - d < c$, we set $v_e = anc(v, k_2)$; otherwise we set $v_e = anc(v', c-1)$. Now we can safely run the tree path query from v_s to v_e.

The other case is that the path lies completely on the cycle, that is, $k_1 \geq d$. We can first exclude the condition $k_2 - k_1 \geq c$, as in this case we simply query the path from v_1 to v_c. If $k_2 - k_1 < c$, we find v_s inside the path that goes from v_1 to v_c: If $depth(v) - k_1 \geq c$, we set $v_s = anc(v, k_1)$; otherwise we set $v_s = anc(v_1, ((c-1)-(depth(v)-k_1)) \bmod c)$. We then do the same to compute v_e with k_2. However, if v_e is deeper than v_s, we recompute $v_e = anc(v_e, c)$. Finally, we run the tree path query from v_s to v_e.

A final issue is how to map back the nodes $u = v'_i$ that the algorithm may return. Note that we know the cycle where the query was performed, so we know c and v_1. Thus, if $depth(u) < c$, we know that u is a created node, and replace it with $anc(v_1, (c-1) - depth(u))$ before mapping it to the domain of f. Algorithm 2 gives the pseudocode.

Since we have up to n newly created nodes for which we have to store labels, we have the following result.

Theorem 2. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a tree representation that performs in constant time the operations id, node, anc, depth, leftmost, and lca, and it addition it solves a certain non-decomposable path query on n-node trees with labels in $[1..\sigma]$ in $T(n,\sigma)$ time, using in total $S(n,\sigma)$ bits of space. Then, there exists a data structure using $n \lg n + O(n) + S(2n+1, \sigma)$ bits that answers the same query on the positive paths of f in time $O(\lg n / \lg\lg n) + T(2n+1, \sigma)$. There exists another data structure using $n \lg n(1+1/t) + O(n) + S(2n+1, \sigma)$ bits that answers the query in time $O(t) + T(2n+1, \sigma)$, for any $t > 0$.*

By considering the selection queries of He et al. [16] (Section 2.3), we obtain the following corollary. Note that, since we duplicate some nodes and their labels, the space of the sequence of labels can grow at most to $2nH(\ell)$ by standard entropy arguments.

Corollary 3. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + 2nH(\ell) + O(n) + o(n \lg \sigma)$ bits that answers selection queries on the positive paths of f in time $O(\lg n / \lg\lg n)$, where $H(\ell) \leq \lg \sigma$*

Algorithm 2: Computing non-decomposable queries on positive paths.

```
1  Proc Positive(i, k1, k2)
2      v ← node(select1(B, π^{-1}(i)))
3      v'_{c-1} ← anc(v, depth(v) − 1)
4      v1 ← anc(leftmost(v'_{c-1}), 1)
5      c ← (depth(v1) + 1)/2
6      v' ← lca(v1, v)
7      d ← depth(v) − depth(v')
8      if k1 < d then
9          vs ← anc(v, k1)
10         if k2 − d < c then ve ← anc(v, k2)
11         else ve ← anc(v', c − 1)
12     else if k2 − k1 ≥ c then
13         vs ← v1
14         ve ← vc
15     else
16         if depth(v) − k1 ≥ c then vs ← anc(v, k1)
17         else vs ← anc(v1, ((c − 1) − (depth(v) − k1)) mod c)
18         if depth(v) − k2 ≥ c then ve ← anc(v, k2)
19         else ve ← anc(v1, ((c − 1) − (depth(v) − k2)) mod c)
20         if depth(vs) < depth(ve) then
21             ve ← anc(ve, c)
22     Compute path query from vs to ve
23     Return the answers; resulting nodes u are converted into domain values
       π(rank1(B, id(u))), but if depth(u) < c we first set
       u ← anc(v1, (c − 1) − depth(u))
```

is the entropy of the distribution of the values in ℓ.

It is also possible to use the variant that requires $n \lg n(1 + 1/t)$ bits, to approach the cost $O(\lg \sigma / \lg \lg \sigma)$ of the base query [16].

3.3 Functions without labels

In the simple case where the function has no assigned labels, or said another way, where we assume $\ell(i) = i$ for the queries, we can do better than Corollaries 1, 2, and 3. In the first case [5], we do not need anymore to store the labels, since these are now simply the elements returned by π. For clarity, we restate the resulting corollary.

Corollary 4. *Let $f : [1..n] \to [1..n]$ be a function. Then, there exists a data*

structure using $n \lg n + O(n)$ *bits that answers minimum/maximum queries on the positive paths of* f *in time* $O(\lg n / \lg \lg n)$*. There exists another structure using* $n \lg n(1 + 1/t) + O(m)$ *bits, for any* $m \geq n$*, that answers the queries in time* $O(t + \alpha(m, n))$*, for any* $t > 0$*. By setting* $t = \alpha(m, n)$*, we obtain a structure using* $n \lg n + o(n \lg n) + O(m)$ *bits that answers the queries in time* $O(\alpha(m, n))$*.*

For Corollaries 2 and 3, all the path query structures [16] store the sequence of labels (now domain elements) in node identifier order, and represent it with a wavelet tree [13]. This structure allows us, with a query similar to *select*, to find the occurrence of element i, thus effectively computing $\pi^{-1}(i)$, in time $O(\lg n / \lg \lg n)$. Instead of returning the node identifier, they may return the label, that is, the domain element, by accessing the wavelet tree in the same time. Therefore, they do not require the permutation to map from elements to nodes. In the case of Corollary 3, where we have duplicated nodes v'_i, we may use the *select*-like operation to find the two places where an element is mentioned in the labels, and choose the one with largest depth to avoid starting the query from a node v'_i. Since the wavelet tree has each distinct element mentioned once or twice, its entropy is essentially maximal, and we have the following results for this case.

Corollary 5. *Let* $f : [1..n] \to [1..n]$ *be a function. Then, there exists a data structure using* $n \lg n + o(n \lg n)$ *bits that answers counting queries on the positive paths of* f *in time* $O(\lg n / \lg \lg n)$*, and also reports those* r *results in time* $O((r + 1) \lg n / \lg \lg n)$*.*

Corollary 6. *Let* $f : [1..n] \to [1..n]$ *be a function. Then, there exists a data structure using* $2n \lg n + o(n \lg n)$ *bits that answers selection queries on the positive paths of* f *in time* $O(\lg n / \lg \lg n)$*.*

4 Negative Path Points

Figure 4 shows the same tree of Figure 2, now showing clearly the resulting levels of the tree, and without the extra root. The result is a forest, which we will store with the Fully-Functional representation [25]. The figure illustrates an important point: all the nodes in $f^{-k}(i)$ correspond to the descendants at distance k of the node corresponding to i. For example $f^{-2}(1) = \{3, 11, 22, 15, 20, 19\}$. These form a range if we deploy the nodes in levelwise order.

Just as for positive paths, we will store a bitvector B indicating which nodes are originally in G (i.e., not the added leaves) and a permutation π on $[1..n]$ mapping from the identifiers of those nodes in G (after being mapped to

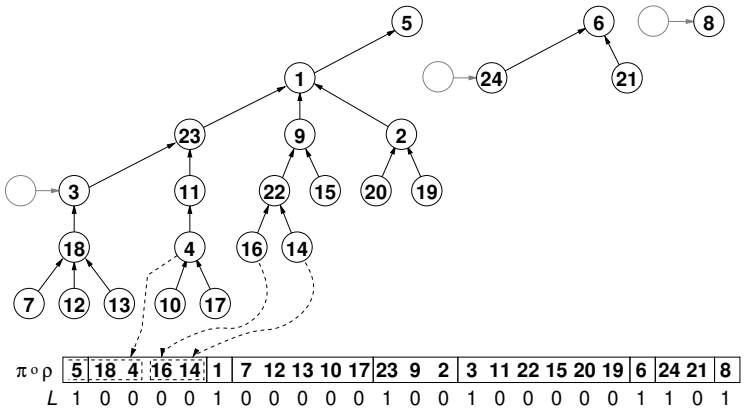

Fig. 4. The same tree used for decomposable positive queries (without the extra root), showing how the levels are deployed to solve negative path point queries. The numbers inside the nodes are the π values of the node identifiers, as before, whereas the dashed arrows represent the ρ^{-1} values of the node identifiers.

$[1..n]$ using B) to domain elements. The information on the nodes (such as the labels) will be stored in levelwise order, with a permutation ρ on $[1..n]$ mapping from the levelwise deployment to the tree identifier of the node (mapped using B). Let $v = \mathit{node}(\mathit{select}_1(B, \pi^{-1}(i)))$ be the node corresponding to domain element i, and assume v is not on the cycle of its component in G. Then the elements of $f^{-k}(i)$ are the descendants of v at distance k. The leftmost such descendant is found with $v_1 = \mathit{fwd}(v, k)$, whereas the rightmost one is $v_2 = \mathit{open}(\mathit{bwd}(\mathit{close}(v), k+1)+1)$. Then the range of values where the information on the elements of $f^{-k}(i)$ is stored is $[\rho^{-1}(\mathit{rank}_1(B, \mathit{id}(v_1))), \rho^{-1}(\mathit{rank}_1(B, \mathit{id}(v_2)))]$. Note that any element at position j in the levelwise deployment can be converted into a domain element with $\pi(\rho(j))$. Figure 4 shows how $f^{-2}(9)$ is mapped to the range containing $(16, 14)$, which is within the level containing $(18, 4, 16, 14)$ (disregard for now bitvector L and the way levels are interlaced in the array).

When v is on a cycle (of length c), then we can go to its predecessor in the cycle (taking the arrow backwards) and collect the descendants at distance $k-1$ in its sprouting tree, then to its predecessor and collect its descendants at distance $k-2$, and so on. Given the way we have converted G into a tree, all these nodes are indeed the descendants of v at distance k; consider again $f^{-2}(1)$ in Figure 4. However, the situation can be more complicated because, if k is large enough, then we could run over the whole cycle in backward direction and return again to v, now looking for descendants at distance $k-c$. Therefore, not only we have to include the descendants of v at distance k, but also all the elements in the whole tree where v belongs at depths $\mathit{depth}(v)+k-c$, $\mathit{depth}(v)+k-2c$, and so on.

To handle this case, we will store the levelwise information on the nodes of each tree of the forest in an interlaced order of the levels: levels 1, $c+1$, $2c+1$,

Algorithm 3: Computing queries on negative path points.

```
1  Proc Negative(i, k)
2      v ← node(select_1(B, π^{-1}(i)))
3      v_c ← anc(v, depth(v))
4      v_0 ← leftmost(v_c)
5      v_2 = open(bwd(close(v), k + 1) + 1)
6      p_2 = ρ^{-1}(rank_1(B, id(v_2)))
7      if v ≤ v_0 ≤ close(v) then
8          p_1 ← select_1(L, rank_1(L, p_2))
9      else
10         v_1 ← fwd(v, k)
11         p_1 ← ρ^{-1}(rank_1(B, id(v_1)))
12     Compute array range query on [p_1, p_2]
13     Return the answers; resulting positions j are converted into domain
       values π(ρ(j)).
```

and so on, then levels 2, $c+2$, $2c+2$, and so on, until levels c, $2c$, $3c$, and so on. A bitvector $L[1..n]$ with *rank* and *select* support will mark, in the levelwise ordered domain, the first node at a level of the form $tc + l$ in each tree, for all $1 \leq l \leq c$. Figure 4 shows the levelwise deployment. The nodes of the first tree are listed as $5, 18, 4, 16, 14$ for $l = 1$, then $1, 7, 12, 13, 10, 17$ for $l = 2$, then $23, 9, 2$ for $l = 3$, and finally $3, 11, 22, 15, 20, 19$ for $l = 4$. The following two trees are then listed as $6, 24, 21$ and 8. The bitvector L marks the beginnings of the change in tree or in l.

With this arrangement, we only have to find as before $p_2 = \rho^{-1}(rank_1(B, id(v_2)))$, the second endpoint of the range, and then $p_1 = select_1(L, rank_1(L, p_2))$, the beginning of the nodes of the tree of v_2 with its same l value. Figure 4 shows how v_2 is found for $f^{-2}(23)$, and then the range includes up to the beginning of $l = 1$ in its tree, to contain $(5, 18, 4)$.

The final issue is how to determine if v is or not on the cycle. We can do this by computing, similarly to the positive paths, $v_c = anc(v, depth(v))$, $v_0 = leftmost(v_c)$ as the leftmost leaf, and then v is in the cycle iff v_0 descends from v, that is, $v \leq v_0 \leq close(v)$.

Finally, we can build on the levelwise deployment of the node data any array range query data structure we desire. Algorithm 3 shows the pseudocode.

The time per query is that of the array range query, plus the time needed to compute π^{-1} and ρ^{-1} a constant number of times; answers are converted back to domain values by computing ρ and π in constant time. Apart from the array range query structures, we are storing two permutations and some bitvectors. We then have the following result.

Theorem 3. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be an array range query data structure that, on an array $A[1..n]$ of values in $[1..\sigma]$, answers queries in time $T(n, \sigma)$ using $S(n, \sigma)$ bits of space. Then, there exists a data structure using $2n \lg n + O(n) + S(n, \sigma)$ bits that answers the same query on the negative path points of f in time $O(\lg n / \lg \lg n) + T(n, \sigma)$. There exists another data structure using $2n \lg n(1 + 1/t) + O(n) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t > 0$.*

4.1 Reducing space

We can also design a more sophisticated variant that uses $n \lg n + o(n \lg n)$ bits, instead of $2n \lg n + O(n)$, by exploiting some regularity present in the mapping ρ. We note that ρ^{-1} maps a tree position $p = \mathit{rank}_1(B, \mathit{id}(v))$ into an array corresponding to the depth of the node, $d = \mathit{depth}(v)$, and an offset inside the array that is equal to the number of nodes with depth v and preorder $\leq p$. The Fully Functional representation offers constant-time operations *prev-level*(v) and *next-level*(v), as we have seen in Section 2.3. We can therefore choose a sampling step t and store, for every level d for which there are t or more nodes of depth d, the position in the array of labels of every tth node of depth d. A bitvector M marks the positions $\mathit{rank}_1(B, \mathit{id}(\cdot))$ of all the marked nodes (for all the values of d), and their corresponding positions in the array of labels is stored in an array P, of at most n/t entries. To map a node v, we compute $v_0 = v$ and $v_{i+1} =$ *prev-level*(v_i), until $M[\mathit{rank}_1(B, \mathit{id}(v_i))] = 1$ for some $0 \leq i < t$. In this case, the position of v is $P[\mathit{rank}_1(M, \mathit{rank}_1(B, \mathit{id}(v_i))) + i]$. We thus compute the mapping ρ^{-1} in time $O(t)$ using $O((n/t) \lg n + n)$ bits of space. Note that we can simply concatenate the arrays M and P of different trees.

An exception occurs when v is the ith node of depth d for some $i < t$, in which case *prev-level*(v_{i-1}) returns a null value. In this case we know i, but we need to know the position where the range of depth $d = \mathit{depth}(v)$ starts in the whole array. This can be computed with operation *select* on a bitvector D analogous to L, which marks the positions where each depth starts. Since the depths are interlaced, the 1 in D corresponding to depth d is computed as follows: let $r = 1 + ((d - 1) \bmod c)$ and $k = 1 + ((h - 1) \bmod c)$, where $h = \mathit{height}(v_c)$ is the height of the tree where v belongs (which is also computed in constant time in the Fully Functional representation). Then, if $r \leq k + 1$, the desired 1 is $d' = (r - 1) \cdot \lceil h/c \rceil + \lceil d/c \rceil$; otherwise we must subtract $r - 1 - k$ from that value. In addition, since the levels of the different trees are separated and concatenated in the array of labels, the ranges of the tree of v start at position $\mathit{rank}_1(B, \mathit{id}(v_c))$, so the global position of v is computed

as $i - 1 + \mathit{select}_1(D, d' + \mathit{rank}_1(D, \mathit{rank}_1(B, \mathit{id}(v_c) - 1)))$.

Once the array range query returns a position j, we must map it back with ρ. We again use D to determine that we are at the range $d' = \mathit{rank}_1(D, j) - \mathit{rank}_1(D, \mathit{rank}_1(B, \mathit{id}(v_c) - 1))$, with offset $i = 1 + j - \mathit{select}_1(D, \mathit{rank}_1(D, j))$. We have another bitvector M' where the sampled positions are aligned to D and marked with a 1, and their corresponding tree nodes are stored in an array P' aligned to the 1s in M'. Thus, the node of the sampled position preceding j is $v' = P'[j']$, with $j' = \mathit{rank}_1(M', j)$. From v', we execute *next-level* for $j - \mathit{select}_1(M', j') < t$ times to obtain the mapping.

Once again, the first positions of each depth are not sampled, which we note because $\mathit{select}_1(M', j') < \mathit{select}_1(D, \mathit{rank}_1(D, j))$. In this case, we must obtain the depth d, the first descendant of v_c at depth d (with $v' = \mathit{fwd}(v_c, d)$), and execute $i - 1$ *next-level* steps from v'. To compute d, we find k as before, compute $r = \lceil d'/\lceil h/c \rceil \rceil$ and $g = (d' - 1) \bmod \lceil h/c \rceil$. However, if $r > k + 1$, we recompute $r = \lceil (d' - k)/\lceil h/c - 1 \rceil \rceil$, $g = (d' - k - 1) \bmod \lceil h/c - 1 \rceil$. Then, we have $d = gc + r$.

We thus obtain the following result. The second variant does the mapping faster and is useful when the array range structure directly returns labels. It is obtained by using for π^{-1} the version that uses $O(n)$ bits and $O(\lg n/\lg\lg n)$ time. We also replace the sampling mechanism used to compute ρ^{-1} by a monotone minimum perfect hash function [1] for each level d, where we store all the values $\mathit{rank}_1(B, \mathit{id}(v))$ of the nodes v for which $\mathit{depth}(v) = d$, and it returns their rank in the set. Such hash function uses $O(n \lg\lg\lg n)$ bits and returns the rank in time $O(\lg\lg n)$. To obtain the offset in the array of level d of the tree where v belongs, we also search for $\mathit{fwd}(v_c, d)$, the first node at level d in its tree, and subtract the ranks. The difference plus 1 is then the offset i in the array of level d of the tree of v. Its starting position in the global array is computed as in the case where we do not have a sampled predecessor node.

Theorem 4. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be an array range query data structure that, on an array $A[1..n]$ of values in $[1..\sigma]$, answers queries in time $T(n, \sigma)$ using $S(n, \sigma)$ bits of space. Then, there exists a data structure using $n \lg n(1 + 1/t) + O(n) + S(n, \sigma)$ bits, for any $t > 0$, that answers the same query on the negative path points of f in time $O(t) + T(n, \sigma)$. If the array query returns array positions that must be converted to domain elements, then each answer requires $O(t)$ additional time. Otherwise, there exists another data structure using $n \lg n + O(n \lg\lg\lg n) + S(n, \sigma)$ bits that answers the query in time $O(\lg n/\lg\lg n) + T(n, \sigma)$.*

By considering the various array range queries of Section 2.4, we obtain the

following corollaries; others can be derived by the reader.

Corollary 7. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n(1 + 1/t) + O(n)$ bits that finds the elements of $f^{-k}(i)$ with the minimum and the maximum labels, for any $i \in [1..n]$ and $k > 0$, in time $O(t)$, for any $t > 0$.*

Corollary 8. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n(1 + 1/t) + O(n \lg R)$ bits, for any $t > 0$ and $R \leq n$. It finds the element with the rth largest label in $f^{-k}(i)$, for any $i \in [1..n]$, $k > 0$, and $1 \leq r \leq R$, in time $O(t + \lg r / \lg \lg n)$. It can also list the r elements with the largest or smallest values in $f^{-k}(i)$ in time $O(tr)$.*

Corollary 9. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n + (1 + \epsilon)nH(\ell) + O(n \lg \lg \lg n)$ bits, where $\epsilon > 0$ is any constant and $H(\ell)$ is the entropy distribution of the labels, that finds the τ-majorities in the labels of $f^{-k}(i)$, for any $i \in [1..n]$, $k > 0$, and $0 < \tau < 1$, in time $O(1/\tau + \lg n / \lg \lg n)$. It can also use $n \lg n(1+1/t)+O(n)+(1+\epsilon)nH(\ell)$ bits, for any $t > 0$, and answer queries in time $O(1/\tau + t)$.*

Corollary 10. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n \lg n(1 + 1/t) + O(n) + nH(\ell) + o(n \lg \sigma)$ bits, for any $t > 0$, where $H(\ell)$ is the entropy distribution of the labels, that counts the number of labels of $f^{-k}(i)$ within a range, for any $i \in [1..n]$, $k > 0$, and range of labels, in time $O(t + \lg \sigma / \lg \lg n)$. It can then list those r elements in time $O(r(t + \lg \sigma / \lg \lg n))$. If only counting is needed, then there is a structure using $n \lg n + nH(\ell) + o(n \lg \sigma) + O(n \lg \lg \lg n)$ bits, that counts in time $O(\lg n / \lg \lg n)$.*

4.2 Functions without labels

As before, we can consider the case where the domain elements have no labels, but we just want to query their values, that is, $\ell(i) = i$. In Corollary 10, the geometric structures used allow performing the equivalent to *select* queries on the sequence of labels, and thus they can be used to find where the domain value i appears in the array of labels, and then we can map it to the tree using ρ and B. Then there is no need for permutation π, and we can

subtract $n \lg n$ bits to the space in this corollary.[6] On the other hand, we have that $H(\ell) = \lg n$ since each symbol appears exactly once in the sequence.

Corollary 11. *Let $f : [1..n] \to [1..n]$ be a function. Then, there exists a data structure using $n \lg n + o(n \lg n)$ bits, that counts the number of elements of $f^{-k}(i)$ within a range, for any $i \in [1..n]$, $k > 0$, and range of elements, in time $O(\lg n / \lg \lg n)$. It can then list those r elements in time $O((r+1) \lg n / \lg \lg n)$.*

5 Negative Paths

For ranges of negative values of k, $f^{-k_1..-k_2}(i)$, our solution maps the queries into two-dimensional ranges, which require more space and/or time than previous solutions. We preserve the same tree as in Section 4, but this time the mapping from node v is done to pairs $(rank_1(B, id(v)), depth(v))$. Here $id(v)$ is the preorder of the node in the Fully-Functional representation. Therefore, once we have mapped the domain element i to a tree node v, and determined that v is not on the cycle, we have that the query encompasses the two-dimensional range $[rank_1(B, id(v)) .. rank_1(B, id(v) + subtreesize(v) - 1)] \times [depth(v) + k_1 .. depth(v) + k_2]$, which is determined in constant time. We now perform the desired query on a structure that handles two-dimensional points (possibly with labels). The returned points (j, d) are then mapped to the domain values $\pi(j)$ in constant time. Note that we can use a single $n \times n$ grid for all the trees.

For the case where v is on the cycle, we will use a further structure. Note that we want to consider, in addition to the previous range, all the nodes in the tree of v with a depth that is between $d_1 = depth(v) + k_1$ and $d_2 = depth(v) + k_2$, modulo c, but not reaching the range $[d_1..d_2]$, as that one is already handled. To this end, we will map the nodes v to pairs $(depth(v) \text{ div } c, depth(v) \bmod c)$, and will query for the points in the range $[0..d_2 \text{ div } c - 1] \times [d_1 \bmod c .. d_2 \bmod c]$. If, however, $d_1 \bmod c > d_2 \bmod c$, then we split the second range into two, $[d_1 \bmod c .. c - 1]$ and $[0 .. d_2 \bmod c]$.

An exception occurs if $k_2 - k_1 \geq c$, since then the two types of ranges overlap and we could count points twice. In this case we take, in this second arrangement, the range $[0 .. d_2 \text{ div } c - 1] \times [0 .. c - 1]$, and reduce the range within the subtree of v to $[rank_1(B, id(v)) .. rank_1(B, id(v) + subtreesize(v) - 1)] \times [(d_2 \text{ div } c) \cdot c .. d_2]$.

Note that all the nodes in a tree with the same depth are mapped to the same

[6] The same happens in Corollary 9, but the query makes no sense if the labels are all unique.

Algorithm 4: Computing queries on negative paths.

```
1  Proc Negative(i, k1, k2)
2      p ← π^{-1}(i)
3      v ← node(select1(B, p))
4      p' ← rank1(B, id(v) + subtreesize(v) − 1)
5      v_c ← anc(v, depth(v))
6      v_0 ← leftmost(v_c)
7      c ← depth(v_0)
8      d1 ← depth(v) + k1
9      d2 ← depth(v) + k2
10     if k2 − k1 > c then
11         Query the first structure with [p .. p'] × [(d2 div c) · c .. d2]
12         Query the second structure with [0..d2 div c − 1] × [0 .. c − 1]
13     else
14         Query the first structure with [p .. p'] × [d1 .. d2]
15         if d1 mod c ≤ d2 mod c then
16             Query the second structure with [0..d2 div c − 1] × [d1 mod c .. d2
               mod c]
17         else
18             Query the second structure with [0..d1 div c − 1] × [d1
               mod c .. c − 1]
19             Query the second structure with [0..d2 div c − 1] × [0 .. d2 mod c]
20     Return the composition of all the points collected in the ranges.
```

cell of this second grid; these are separated in the geometric structures with standard methods, in which the width of the grid is enlarged to have one point per column. The grids of all the trees can then be horizontally concatenated, to reach n columns in total. The strip of the grid for the node v is then the one starting at column $rank_1(B, id(v_c))$, as before. Finally, note that we must spend other $n \lg n$ bits to maintain the domain values associated with the returned nodes in this second grid, since this time they cannot be deduced only from the depth information stored at the points.

Note that we have to complete the query from the results of up to 3 two-dimensional ranges, so the query must be decomposable. Algorithm 4 gives the details.

Theorem 5. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a two-dimensional range query data structure that, on an $n \times n$ grid with n points with values in $[1..\sigma]$, answers decomposable queries in $T(n, \sigma)$ time using in $S(n, \sigma)$ bits of space. Then, there exists a data structure using $2n \lg n + O(n) + 2S(n, \sigma)$ bits that answers the same query on the negative paths of f in time $O(\lg n / \lg \lg n) +$*

$T(n,\sigma)$. There exists another data structure using $2n\lg n(1+1/t)+O(n)+2S(n,\sigma)$ bits that answers the query in time $O(t)+T(n,\sigma)$, for any $t>0$.

We can combine this theorem with various results on querying two-dimensional grids of points with labels (or weights); recall Section 2.5. The other queries considered in that section are not decomposable.

Corollary 12. *Let $f:[1..n]\to[1..n]$ be a function and $\ell:[1..n]\to[1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $O(n\lg n)$ bits that answers minima and maxima queries on the negative paths of f in time $O(\lg^{1+\epsilon} n)$, and top-r queries in time $O((r+\lg n)\lg^{\epsilon} n)$, for any constant $\epsilon>0$.*

Corollary 13. *Let $f:[1..n]\to[1..n]$ be a function and $\ell:[1..n]\to[1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $O(n\lg^2 n/\lg\lg n)$ bits that answers range counting queries on the negative paths of f in time $O((\lg n/\lg\lg n)^2)$, and reports the r values in time $O((r+1)(\lg n/\lg\lg n)^2)$. By using slightly more space, $O(n\lg^{2+\epsilon} n)$ bits for any constant $\epsilon>0$, the time to report is reduced to $O(r+\lg\lg n)$.*

6 Navigation Queries

We now consider a different type of query, which allow us navigate on the graph of the function in a more sophisticated way than the basic iteration queries of Munro et al. [21]. We study various queries of this type that might be of interest.

6.1 Nearest common successor

Given $i,i'\in[1..n]$, we want to know which is the element j such that $f^k(i)=f^{k'}(i')=j$ for the minimum possible values of $k\geq 0$ and $k'\geq 0$. This is the closest "meeting point" of i and i' through applications of f. Note that there may be more than one solution not dominating each other.

It is not hard to see that this query can be solved with the arrangement of Section 3.1. We first compute the corresponding nodes, $v=node(select_1(B,\pi^{-1}(i)))$ and $v'=node(select_1(B,\pi^{-1}(i')))$. Now, if v and v' are in the same subtree sprouting from the node of a cycle, then the only answer is $j=\pi(rank_1(B,id(w)))$, where $w=lca(v,v')$. If w is the (fake) tree root, then v and v' are in different trees, and there is no answer to the query.

Algorithm 5: Computing the meeting point of two values.

```
1  Proc Meeting(i, i')
2      v ← node(select_1(B, π^{-1}(i)))
3      v' ← node(select_1(B, π^{-1}(i')))
4      w ← lca(v, v')
5      if w = root then
6          Return that there is no answer
7      v_c ← anc(v, depth(v) − 1)
8      v_0 ← leftmost(v_c)
9      u ← lca(v_0, v)
10     u' ← lca(v_0, v')
11     if u = u' then
12         Return one answer: π(rank_1(B, id(w)))
13     else
14         Return two answers: π(rank_1(B, id(u))) and π(rank_1(B, id(u')))
```

The other case is that v and v' are in the same tree, but they come from subtrees sprouting from different nodes of the cycle. This case yields two incomparable answers. We first compute the leftmost leaf v_0 of the cycle. Then, the points where v and v' meet the cycle, $u = lca(v_0, v)$ and $u' = lca(v_0, v')$, are valid answers: the first with $k = depth(v) - depth(u)$ and $k' = depth(v') - depth(u') + ((depth(u') - depth(u)) \bmod c)$; and the second with $k' = depth(v') - depth(u')$ and $k = depth(v) - depth(u) + ((depth(u) - depth(u')) \bmod c)$. Algorithm 5 shows the procedure.

By using the Fully-Functional tree representation [25], we obtain the following result.

Theorem 6. *Let $f : [1..n] \to [1..n]$ be a function. Then, there exists a data structure using $n \lg n + O(n)$ bits that finds the nearest common successor of two values in time $O(\lg n / \lg \lg n)$. There exists another data structure using $n \lg n(1 + 1/t) + O(n)$ bits that answers the query in time $O(t)$, for any $t > 0$.*

A related query is whether there exists some $k \geq 0$ such that $f^k(i) = f^k(i')$, and which is the smallest such k. This is easily solved after finding the meeting point(s) in Algorithm 5. If there is only the meeting point $w = lca(v, v')$, then the answer is $depth(v) - depth(w)$ if this is equal to $depth(v') - depth(w)$; otherwise there is no answer. If, on the other hand, we find two answers u and u', each with its own values k and k' (which are not computed in Algorithm 5, but we gave their formulas), then an answer is k if $k = k'$. Then we may return the k of u or the k of u', depending on which is smaller.

Given $i \in [1..n]$ and $1 \leq \ell_1 \leq \ell_2 \leq \sigma$, this query asks for the smallest $k \geq 0$ such that $\ell(f^k(i)) \in [\ell_1, \ell_2]$. We can answer this query by using the arrangement of Section 3.2 and building on the resulting trees a structure that finds the nearest ancestor with a label in a given range.

We define, over a labeled tree, the operator *lowest-anc*$_{[\ell_1..\ell_2]}(T, x)$, which returns the lowest ancestor of x with a label in $[\ell_1..\ell_2]$ if it exists, or null otherwise. We first show how to support *lowest-anc* in constant time when $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1/2$.

The starting point of our solution is the succinct tree representations based on tree covering [12,15,8]. Given a parameter M, Geary et al. [12] showed how to cover all the nodes in T using a set of $O(n/M)$ *cover elements*, each being a subtree of size $O(M)$. Any two cover elements are either disjoint (i.e., not sharing any nodes), or only share the common root. Farzan and Munro [8] further modified this approach to guarantee that, for each cover element, there exists at most one edge that connects a non-root node of this cover element to a node of a different element. Using their approach, the number of nodes in each cover element is bounded by $2M$. Succinct representations of (unlabeled) trees can be constructed by constructing auxiliary data structures for two tiers of cover elements of T [12,15,8].

In our solution, we first follow the approach of representing a labeled tree over a small alphabet by He et al. [17], and then build additional auxiliary data structures to support *lowest-anc*. In their approach, He et al. adopt the idea by Geary et al. [12] to represent the tree structure of T, but with different parameters and a different approach of computing cover elements. More precisely, we first set $M = \lceil \lg^2 n \rceil$ and use the approach of Farzan and Munro to cover T with *mini-trees* of size at most $2M$. Then, for each mini-tree, we set $M' = \lceil \lg^\lambda n \rceil$ for some $\max(1/2, 2\epsilon) < \lambda < 1$ to cover it with micro-trees of size at most $2M'$.[7] Mini-trees (or micro-trees) are ordered by the preorder of their roots, and when two mini-trees (or micro-trees) share the same root, they are ordered by the immediate successor of the root in these trees in preorder. Thus, when referring to a node of T, there are two ways: Externally, we identify each node by its preorder. Internally, we define the τ-name of a node x to be $\tau(x) = \langle \tau_1(x), \tau_2(x), \tau_3(x) \rangle$, where $\tau_1(x)$ is the rank of the mini-tree containing x, $\tau_2(x)$ is the rank of the micro-tree containing x among all the micro-trees in the $\tau_1(x)$-th mini-tree, and $\tau_3(x)$ is the preorder rank of x in

[7] He et al. [17] assume $0 < \epsilon < 1$ and $\max(1/2, \epsilon) < \lambda < 1$. To support *lowest-anc*, we have to further narrow down the ranges of possible values for ϵ and λ. Since the ranges set by us are subranges of those used by He et al., their data structures can still be directly used in our representation.

its micro-tree. We also say that $\tau_i(x)$ is the τ_i-name of x, for $i \in \{1, 2, 3\}$. As He et al. pointed out, the auxiliary data structures of Geary et al. [12] and He et al. [15] can be easily adapted to our parameters to represent the structure of T (without labels) in $2n + o(n)$ bits such that:

- We can perform the conversion between the preorder rank and the τ-name of an arbitrary node in constant time. Thus, we can refer to a node by either its preorder rank or its τ-name, without explicitly mentioning the conversion process.
- Given the τ-name of a node x, we can compute the global rank of the micro-tree containing x among all micro-trees of T in constant time.
- We can also support a list of (unlabeled) operations in constant time. The operations that are used in our solution include computing the depth and the degree of x, and locating the i-child of x for any given i.

To support operations over the labels of the nodes of T, we encode the labels as a sequence ordered by the preorder ranks of the nodes and build the same auxiliary data structures as those designed by He et al. [17]. The bounds of their solution can be summarized in the following lemma, in which an *α-node* is defined to be a node labeled α.

Lemma 1 ([17])**.** *Let T be an ordinal tree on n nodes, each assigned a label from $[1..\sigma]$, where $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1/2$. Then T can be represented in $nH(\ell) + 2n + o(n)$ bits, where $H(\ell)$ is the entropy distribution of the labels, to support the following operations in $O(1)$ time:*

- *pre-rank$_\alpha(T, x)$, which returns the number of α-nodes preceding x in preorder;*
- *pre-select$_\alpha(T, i)$, which returns the i-th α-node in preorder;*
- *pre-count$_\beta(T, i)$, which returns the number of nodes that have preorder ranks at most i and labels at most β;*
- *parent$_\alpha(T, x)$, which returns the lowest ancestor of x that is labeled α.*

In addition, the encoding of an arbitrary micro-tree of T, including the tree structure and the node labels, can be computed in $O(1)$ time.

Note that more operations are supported by He et al. [17], but we only use those defined in this lemma. We now construct additional auxiliary structures to support *lowest-anc*.

Lemma 2. *With additional auxiliary structures of $o(n)$ bits, the data structures in Lemma 1 can support lowest-anc in $O(1)$ time.*

Proof. Let n_1 and n_2 denote the numbers of mini- and micro-trees, respectively. We construct the following auxiliary data structures:

- A two-dimensional array $A[1..n_1, 1..\sigma, 1..\sigma]$, in which $A[i, \ell_1, \ell_2]$ stores the preorder number of the node *lowest-anc*$_{[\ell_1..\ell_2]}(T, r_i)$, where r_i is the root of the i-th mini-tree, or -1 if such a node does not exist;
- A two-dimensional array $B[1..n_2, 1..\sigma, 1..\sigma]$, in which $B[i, \ell_1, \ell_2]$ stores the τ_2- and τ_3-names of the node *lowest-anc*$_{[\ell_1..\ell_2]}(T, r'_i)$, where r'_i is the root of the i-th micro-tree, or -1 if such a node does not exist;
- A table C that stores, for every possible labeled tree L on at most $2M'$ nodes with labels drawn from $[1..\sigma]$, every integer x in $[1..2M']$, and every pair of labels, ℓ_1 and ℓ_2, in $[1..\sigma]$, the preorder number of node *lowest-anc*$_{[\ell_1..\ell_2]}(L, x)$, or -1 if such a node does not exist.

We now analyze the space cost. A occupies $n_1 \cdot \sigma^2 \cdot O(\lg n) = O(n/\lg^{1-2\epsilon} n) = o(n)$ bits, since $n_1 = O(n/\lg^2 n)$ and each entry can be stored in $O(\lg n)$ bits. By the tree covering algorithm, there are $O(\lg^{2-\lambda})$ micro-trees in any mini-tree, and each micro-tree has at most $2\lg^\lambda n$ nodes. Therefore, each entry of B, which consists of a τ_2-name and a τ_3-name only, can be stored in $O(\lg\lg n)$ bits. We also have $n_2 = O(n/\lg^\lambda n)$. Thus, B occupies $n_2 \cdot \sigma^2 \cdot O(\lg\lg n) = O(n \lg\lg n/\lg^{\lambda-2\epsilon} n) = o(n)$ bits. As each micro-tree has at most $2M'$ nodes and entry entry of table C can be encoded in $O(\lg\lg n)$ bits, table C uses at most $2^{4M'} \cdot \sigma^{2M'} \cdot 2M' \cdot \sigma^2 \cdot O(\lg\lg n) = 2^{2\lceil\lg^\lambda n\rceil(\lg\sigma+2)} \cdot 2\lceil\lg^\lambda n\rceil \cdot \sigma^2 \cdot O(\lg\lg n) = o(n)$ bits. Therefore, these auxiliary data structures occupy $o(n)$ bits in total.

To compute *lowest-anc*$_{[\ell_1..\ell_2]}(T, x)$, we first compute the encoding of the micro-tree containing x and then perform a table lookup in C to find out if the answer is inside the same micro-tree that contains x. If it is, this lookup also gives us the answer when combined with the τ_1- and τ_2-names of x, which we return. If x does not have an ancestor in its micro-tree that has a label in $[\ell_1..\ell_2]$, we compute the rank, j, of the micro-tree containing x. The entry $B[j, \ell_1, \ell_2]$ tells us whether the answer is located in the same mini-tree that contains x. If it is, we retrieve the τ_2- and τ_3-names of the answer from this entry, and combine it with $\tau_1(x)$ to compute the answer and then we are done. If not, the entry $A[\tau_1(x), \ell_1, \ell_2]$ will either give us the answer, or tell us that no ancestor of x has a label in $[\ell_1..\ell_2]$. All these steps require $O(1)$ time in total. □

We then make use of the *tree extraction* approach of He et al. [17] to extend Lemma 2 to general alphabets. Given a range, $[\ell_1, \ell_2]$ of labels, an ordinal forest F_{ℓ_1,ℓ_2} can be constructed from a labeled tree T via tree extraction as follows: We remove each non-root node x whose label is not within the range $[\ell_1, \ell_2]$ from T in an arbitrary order and insert its children in place of x into the list of children of its parent, preserving the original left-to-right order. Afterward, if the the label of the root is not within the range, we remove it from the tree, and we do nothing otherwise. The resulting forest is F_{ℓ_1,ℓ_2}.

To represent T for an arbitrary σ, we follow the approach of He et al. [17], and augment the resulting data structure with the auxiliary structures of

Lemma 2. More precisely, we define a conceptual range tree on $[1..\sigma]$ with branching factor $q = \lceil \lg^{\epsilon} n \rceil$. Each node of this tree represents a subrange of the alphabet, and the root corresponds to the entire range of alphabet, $[1..\sigma]$. This tree can be constructed level by level, starting from the top. At each level, for a node representing range $[a..b]$, we partition its range into q subranges, $[a_1..b_1], [a_2..b_2], \ldots, [a_q..b_q]$, such that $a_i = b_{i-1}+1$ for $i = 2, 3, \ldots, q$ and the size of each range, with the possible exception of the last one, is $\lceil (b - a + 1)/q \rceil$. The i-th child of this node then represents $[a_i..b_i]$. We keep repeating this process until all the nodes in the current level represent either empty ranges or ranges of size 1. We number the levels of this range tree $1, 2, \ldots, h$ starting from the root level, where h is the height of this tree. Then $h = O(1 + \lg \sigma / \lg \lg n)$.

For each level, l, of the range tree except the leaf level, we construct an auxiliary tree T_l as follows. Let $[a_1..b_1], [a_2..b_2], \ldots, [a_m..b_m]$ be the nonempty ranges represented by the nodes at the l-th level from left to right. Then $a_1 = 1$, $b_m = \sigma$, and $a_i = b_{i-1} + 1$ for $i = 2, 3, \ldots, m$. T_l has a dummy root, whose children, from left to right, are roots of the trees in $F_{a_1,b_1}, F_{a_2,b_2}, \ldots, F_{a_m,b_m}$, and for the trees from the same forest F_{a_i,b_i}, their left-to-right order is preserved when inserting their root nodes as the children of the dummy root. The dummy root is not taken into account for the preorder sequence or node depths of T_l. We assign labels to nodes of T_l as follows: The dummy root of T_l is assigned label 1. For each node x of T, let x_l denote the node in T_l that corresponds to x. We assign label α to x_l, if the range containing the label of x at the $(l + 1)$-st level of the range tree is the α-th child range of that containing the same label at the l-th level. Thus T_l is a labeled tree on $n + 1$ nodes, each assigned a label from $[1..q]$, where $q = \lceil \lg^{\epsilon} n \rceil$. Each T_l is then represented by the data structures for Lemmas 1 and 2. By the analysis of He et al. [17], all the T_l's together occupy $nH(\ell) + O(n(1 + \lg \sigma / \lg \lg n))$ bits, where $H(\ell)$ is the entropy distribution of the labels.

To support the operation *lowest-anc*$_{[\ell_1..\ell_2]}(T, x)$, we design a recursive algorithm, which is presented in Algorithm 6. In this algorithm, provided that $[a..b]$ is a range at the l-th level of the conceptual range tree, c and d are the numbers of non-root nodes in T_l with labels less than or equal to $a - 1$ and b, respectively, and v is the node in T_l that corresponds to the lowest ancestor of x with a label in $[a..b]$, the procedure $Ancestor([a..b], l, c, d, v, [\ell_l..\ell_2])$ returns the preorder number of the node in T_l that corresponds to the lowest ancestor of x with a label in $[a..b] \cap [\ell_1..\ell_2]$. Thus, to compute *lowest-anc*$_{[\ell_1..\ell_2]}(T, x)$, the initial call to this recursive algorithm is $Ancestor([1..\sigma], 1, 0, n, x, [\ell_l..\ell_2])$.

In the procedure *Ancestor*, we first identify, in lines 4 and 5, the smallest subscript α and the largest subscript β such that the child ranges $[a_\alpha..b_\alpha], [a_{\alpha+1}..b_{\alpha+1}], \ldots [a_\beta..b_\beta]$ are completely contained in $[a^*, b^*]$. Then the return value of this procedure is the lowest node in T_l among these three can-

Algorithm 6: The recursive algorithm for computing the lowest ancestor of v with label in $[\ell_1..\ell_2]$. We assume that *parent* returns a fake node $\perp$ if there is no answer. Node $\perp$ is assumed to be higher than any other node.

```
1  Proc Ancestor([a..b], l, c, d, v, [ℓ1..ℓ2])
2      Let [a1..b1], [a2..b2], ..., [aq..bq] be the child ranges of [a..b] in the range tree
3      [a*..b*] ← [a..b] ∩ [ℓ1..ℓ2]
4      α ← min{i ∈ [1..q], a* ≤ ai ≤ bi}
5      β ← max{i ∈ [1..q], ai ≤ bi ≤ b*}
6      u ←⊥
7      if α ≤ β then
8          u ← lowest-anc_[α..β](Tl, v)
9      for γ ∈ {α − 1, β + 1} do
10         if 1 ≤ γ ≤ q and [aγ..bγ] ∩ [a*..b*] ≠ ∅ then
11             cγ ← c + pre-count_{γ−1}(Tl, d) − pre-count_{γ−1}(Tl, c)
12             dγ ← c + pre-count_γ(Tl, d) − pre-count_γ(Tl, c)
13             pl ← parent_γ(Tl, v)
14             if pl ≠⊥ then
15                 pl+1 ← c + pre-rank_γ(Tl, pl) − pre-rank_γ(Tl, c)
16                 u'l+1 ← Ancestor([aγ..bγ], l + 1, cγ, dγ, pl+1, [ℓ1..ℓ2])
17                 if u'l+1 ≠⊥ then
18                     u'l ← pre-select_γ(Tl, pre-rank_γ(Tl, c) + u'l+1 − cγ)
19                     u ← the lower of u and u' in Tl
20     Return u
```

didate nodes: the lowest ancestor of x in T_l with a label in $[a_{\alpha-1}..b_{\alpha-1}] \cap [\ell_1..\ell_2]$, the lowest ancestor of x in T_l with a label in $[a_\alpha..b_\beta] \cap [\ell_1..\ell_2]$, and the lowest ancestor of x in T_l with a label in $[a_{\beta+1}..b_{\beta+1}] \cap [\ell_1..\ell_2]$. Note that not all three candidate nodes exist necessarily. The second candidate node can be located by performing *lowest-anc* over T_l if it exists (lines 7 and 8). This operation can be supported in constant time by Lemma 2. The first and third candidates are located in lines 9–19. To show how these lines work, take the first candidate for example. The key idea is to recursively perform the same procedure at level $l+1$, and line 16 shows the recursive call. Before this call, we fix the arguments to pass to it. The computation of these arguments is straightforward, except that of p_{l+1}, which is the node in T_{l+1} that corresponds to the lowest ancestor of x with a label in $[a_{\alpha-1}..b_{\alpha-1}]$. To compute p_{l+1}, we first compute p_l, which is the node in T_l that corresponds to T_{l+1} in line 13, and then convert it to p_{l+1} in line 15. After the recursive call, we located the node, u'_{l+1}, in T_{l+1}, that corresponds to the first candidate, and we locate the candidate itself in T_l in line 18.

To analyze the running time, the key observation is that each call to *Ancestor* invokes at most two recursive calls, and furthermore, the only possible case in which two recursive calls are made is that the lowest common ancestor of the leaves in the range tree corresponding to labels ℓ_1 and ℓ_2 are at level l. Therefore, the recursive procedure is called at most $2h$ times. In each call, a constant number of operations are performed at T_l, which require $O(1)$ time in total by Lemmas 1 and 2, as well as the constant-time support for the unlabeled operations listed before them. Therefore, the total time needed to support *lowest-anc* is $O(h) = O(1 + \lg\sigma/\lg\lg n)$. We have proved the following theorem.

Theorem 7. *Let T be a tree on n nodes with labels in $[1..\sigma]$. Then T can be represented in $nH(\ell) + O(n(1 + \lg\sigma/\lg\lg n))$ bits, where $H(\ell)$ is the entropy distribution of the labels, to support lowest-anc in $O(1 + \lg\sigma/\lg\lg n)$ time.*

By applying this technique over the arrangement of Section 3.2, we obtain an analogous result on functions.

Theorem 8. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n\lg n + 2nH(\ell) + O(n) + o(n\lg\sigma)$ bits, where $H(\ell)$ is the entropy distribution of the labels, that answers nearest labeled successor queries in time $O(1 + \lg\sigma/\lg\lg n)$. There exists another data structure using $n\lg n(1 + 1/t) + 2nH(\ell) + O(n) + o(n\lg\sigma)$ bits that answers queries in time $O(t + \lg\sigma/\lg\lg n)$, for any $t > 0$.*

With this query and the previous one we can find the *nearest common labeled successor*: given $i, i' \in [1..n]$, find the element j such that $f^k(i) = f^{k'}(i') = j$ for the minimum possible values of $k \geq 0$ and $k' \geq 0$, but subject to the condition that $\ell(j) \in [\ell_1, \ell_2]$: We first compute the unlabeled answer(s) j with Algorithm 5, and then find the nearest labeled successor of each. The answer closest to i or to i' is the result.

Another generalization is the *iterated nearest successor*, which given an additional parameter i, returns the iteration of the nearest common ancestor query i times. For the particular case of ranges of the form $[\ell_1, \ell_2] = [\ell, \ell]$, we can efficiently answer this query by representing T with the arrangement of Tsur [26], which solves the *labeled level ancestor* query on trees in time $O(\lg \frac{\lg\sigma}{\lg w})$ on a computer word of $w = \Omega(\lg n)$ bits, using $H_\ell + o(n\lg\sigma) + O(n)$ bits. We then obtain the same space of Theorem 8 for this query.

An analogous query, now referring to negative powers of f, is, given $i \in [1..n]$ and $1 \leq \ell_1 \leq \ell_2 \leq \sigma$, find the smallest $k \geq 0$ such that there exists a $j \in f^{-k}(i)$ such that $\ell(j) \in [\ell_1, \ell_2]$.

This query can be solved as a two-dimensional range minimum query. Analogously as in Section 5, we map each node v to the point $(rank_1(B, id(v)), \ell(\pi(rank_1(B, id(v)))))$, with value $depth(v)$. At query time, we map i to its tree node v. If v is not on the cycle, we simply query for the two-dimensional range $[rank_1(B, id(v)) \,..\, rank_1(B, id(v) + subtreesize(v) - 1)] \times [\ell_1, \ell_2]$, and find the node with the minimum depth. If the resulting depth is d, then the minimum distance to v is $k = d - depth(v)$, and we finish.

If v is on the cycle, then any node in the tree rooted at v_c is also a possible candidate, because it is reacheable through the cycle. In this case, we obtain a second candidate answer by querying the whole range $[rank_1(B, id(v_c)) \,..\, rank_1(B, id(v_c) + subtreesize(v_c) - 1)] \times [\ell_1, \ell_2]$. If we find a node with minimum depth d, then it is at distance $d + depth(v_0) - depth(v) - 1$ from v (if it is not already a descendant of v, in which case it will be found, with a shorter distance, with the method of the previous paragraph). We then choose, between the two answers, the one that is closer to v.

By using the structure for two-dimensional minimum range queries (Section 2.5), we obtain the following result.

Theorem 9. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $O(n \lg n)$ bits that answers nearest labeled predecessor queries on f in time $O(\lg^{1+\epsilon} n)$, for any constant $\epsilon > 0$.*

7 Conclusions

Munro et al. [21] studied how to represent an integer function $f : [1..n] \to [1..n]$ so as to efficiently find any element reachable through positive and negative powers of f. We have now considered, for the first time, summary queries on ranges of positive or negative powers of f. For positive powers, we essentially retain optimal storage space and almost match the best results of path queries on trees [16,5]. Negative powers lead to a set of domain values. For a single negative power, we still almost retain the performance of the corresponding array range query, which is generally better than for tree path queries. For a range of negative powers, we resort to two-dimensional range queries, where

the space might stay linear or, if labels are also involved, it might be multiplied by $O(\lg n)$. We also studied other navigation queries through paths of f, more sophisticated than merely iterating positive and negative powers.

This is the first study on this problem, and it is not clear whether the results can be improved, in particular it is not clear if queries on ranges of negative powers of f must resort to three-dimensional range queries. Another interesting line of work is to devise generalizations of integer functions that can be dealt with efficiently.

Acknowledgements

We thank Srinivasa Rao Satti for useful discussions.

References

[1] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16(3):article 2, 2011.

[2] D. Belazzougui, T. Gagie, J. I. Munro, G. Navarro, and Y. Nekrich. Range majorities and minorities in arrays. *CoRR*, abs/1606.04495, 2016.

[3] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[4] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. 11th International Symposium on Algorithms and Data Structures (WADS)*, LNCS 5664, pages 98–109, 2009.

[5] T. M. Chan, M. He, J. I. Munro, and G. Zhou. Succinct indices for path minimum, with applications to path reporting. In *Proc. 22th Annual European Symposium on Algorithms (ESA)*, LNCS 8737, pages 247–259, 2014.

[6] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[7] D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[8] A. Farzan and J. I. Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.

[9] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 371(1):115–121, 2007.

[10] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[11] T. Gagie, M. He, and G. Navarro. Path queries on functions. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LIPIcs 78, page article 5, 2017.

[12] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[13] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[14] R. Grossi, J. Iacono, G. Navarro, R. Raman, and R. Rao Satti. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Transactions on Algorithms*, 13(2):article 28, 2017.

[15] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms*, 8(4):article 42, 2012.

[16] M. He, J. I. Munro, and G. Zhou. Succinct data structures for path queries. In *Proc. 20th Annual European Symposium on Algorithms (ESA)*, LNCS 7501, pages 575–586, 2012.

[17] M. He, J. Ian Munro, and G. Zhou. Data structures for path queries. *ACM Transactions on Algorithms*, 12(4):53:1–53:32, 2016.

[18] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[19] J. Jájá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multi-dimensional dominance reporting and counting. In *Proc. 15th International Symposium on Algorithms and Computation (ISAAC)*, pages 558–568, 2004.

[20] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[21] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[22] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[23] G. Navarro and Y. Nekrich. Time-optimal top-k document retrieval. *SIAM Journal on Computing*, 46(1):89–113, 2017.

[24] G. Navarro, Y. Nekrich, and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[25] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[26] D. Tsur. Succinct representation of labeled trees. *Theoretical Computer Science*, 562:320–329, 2015.