Path Queries on Functions*

Travis Gagie[1,2], Meng He[3], and Gonzalo Navarro[1,4]

1 **CeBiB – Center for Biotechnology and Bioengineering**
2 **School of Computer Science and Telecommunications, Diego Portales University, Chile.** `travis.gagie@gmail.com`
3 **Faculty of Computer Science, Dalhousie University, Canada.** `mhe@cs.dal.ca`
4 **Dept. of Computer Science, University of Chile, Chile.** `gnavarro@dcc.uchile.cl`

Abstract

Let $f : [1..n] \rightarrow [1..n]$ be a function, and $\ell : [1..n] \rightarrow [1..\sigma]$ indicate a label assigned to each element of the domain. We design several compact data structures that answer various queries on the labels of *paths* in f. For example, we can find the minimum label in $f^k(i)$ for a given i and any $k \geq 0$ in a given range $[k_1..k_2]$, using $n \lg n + O(n)$ bits, or the minimum label in $f^{-k}(i)$ for a given i and $k > 0$, using $2n \lg n + O(n)$ bits, both in time $O(\lg n / \lg \lg n)$. By using $n \lg \sigma + o(n \lg \sigma)$ further bits, we can also count, within the same time, the number of labels within a range, and report each element with such labels in $O(1 + \lg \sigma / \lg \lg n)$ additional time. Several other possible queries are considered, such as top-t queries and τ-majorities.

1998 ACM Subject Classification E.1 Data Structures; E.4 Coding and Information Theory

Keywords and phrases Succinct data structures; Integer functions; Range queries; Trees and permutations

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

We focus on the representation of *integer functions* where the domain coincides with the image, $f : [1..n] \rightarrow [1..n]$. This kind of functions were studied by Munro et al. [10], 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 $n \lg n + O(n)$ bits so that any positive power $f^k(i)$ is computed in time $O(1)$, and any negative power $f^{-k}(i)$ is listed in time $O(|f^{-k}(i)|)$. 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 $0 < k_1 \leq k_2$.

* Supported with Basal Funds FB0001, Conicyt, Chile, and by Fondecyt grant 1-140796, Chile.

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 elements are assigned a label $\ell : [1..n] \rightarrow [1..\sigma]$, and we perform summary queries on the labels. 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 the element of P with the rth smallest or largest label, including queries where the value of r is relative to $|P|$ such as median queries. (3) *Top-t queries:* Return a set $M \subseteq P$ formed by t elements with smallest or largest labels in P. (4) *τ-Majority queries:* Return a set of labels whose relative frequency in P 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, e.g., 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 statistics 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 easiest. We build on the recent results of He et al. [8] and Chan et al. [3], who give succinct (and also larger) structures for various path queries on trees. Then a relatively simple unfolding 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, we can solve minimum or maximum queries using $n \lg n + O(n)$ bits and $O(\lg n / \lg \lg n)$ time, range queries in $n \lg n + n \lg \sigma + O(n) + o(n \lg \sigma)$ bits and $O(\lg n / \lg \lg n)$ time per returned element, and selection queries in $n \lg n + 2n \lg \sigma + O(n) + o(n \lg \sigma)$ bits and $O(\lg n / \lg \lg n)$ time.

For negative path points, we unroll the cycles in 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. Since array ranges are particular cases of tree paths, all the complexities obtained for positive paths are inherited by negative path points, but in addition we can perform other queries that have good solutions on array ranges. For example, we can solve top-t queries [13] using $n \lg n + O(n \lg T)$ bits, where T is the maximum t value permitted, in time $O(t + \lg n / \lg \lg n)$. 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 $O(\lg n / \lg \lg n + 1/\tau)$ [1].

The hardest case is the general negative paths. Our queries in this case are mapped into a three-dimensional space, and thus the structures require $O(n \lg n)$ space in order to offer polylogarithmic times. Still, there is no previous result in this case, and thus it is left open whether those queries can be solved efficiently within linear space.

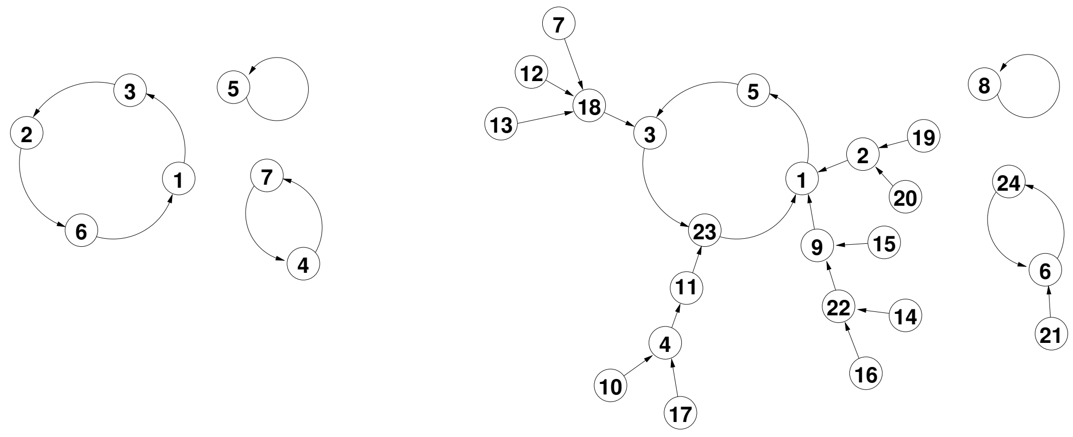

Figure 1 On the left, the representation of a permutation as a directed graph. On the right, the permutation is extended into a function.

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 [5]. 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.

2.2 Permutations and functions

Munro et al. [10] 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 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. 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 $t \le \lg n$, 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 $k \ge 0$, 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 $n \lg n(1 + 1/t)$ bits of space, for any $t \le \lg n$.

2.3 Path queries on trees

He et al. [8] and Chan et al. [3] 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.[1] Given a 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, among others:

1. Minimum/maximum queries, that is, find 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 [3].
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 [8].
3. Range queries include counting, that is, how many nodes in P have labels in $[\ell_1..\ell_2]$, and reporting, that is, reporting 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 $O((r + 1)(1 + \lg \sigma / \lg \lg n))$ [8]. By using more space, it is possible to match the same results of two-dimensional range queries [3].

Those structures include an $O(n)$-bit representation of the tree topology. There are several alternatives (see [15, Ch. 8]) using $2n + o(n)$ bits and supporting a wide set of navigation operations on trees. For positive paths, it turns out that the representations for the path queries used in this section [8, 3] support in constant time a few queries that will be useful:

- 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 (FF) representation [14], 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. Within $2n + o(n)$ bits it supports in constant time all of the above operations, plus $fwd(x, d)$ and $bwd(x, d)$, defined as follows. Let $excess(y)$ be the number of opening minus closing parentheses up to position y in the parentheses sequence. Then $fwd(x, d)$ (resp. $bwd(x, d)$) finds the closest position y to the right (resp. to the left) of 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. $open(x) = bwd(x, 0) + 1$).

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 [6]. An analogous result holds for maximum queries.

[1] 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.

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 $O(1 + \lg r / \lg\lg n)$ [13]. Note that we can set $R = n$ for maximum generality.
3. Top-t queries, that is, finding t elements in $A[i..j]$ with largest labels, can be answered in optimal time $O(t)$ with a structure that uses $O(n \lg T)$ bits and does not access A, where T is an upper bound on the values of t that can be queried [13].
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 $\omega(1/\tau)$ [1].
5. Range counting can be performed in $O(1 + \lg\sigma / \lg\lg n)$ time, and reporting of the r results can be done in time $O((r+1)(1 + \lg\sigma/\lg\lg n))$, using $n \lg\sigma + o(n \lg\sigma)$ bits [2].

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-t elements in a two-dimensional range of an $n \times n$ grid with points having weights in $[1..\sigma]$ can be computed in time $O((t + \lg n) \lg^{\epsilon} n)$, for any constant $\epsilon > 0$, with a data structure that uses $O(n \lg n)$ bits [11, Lem. 7.1]. With $t = 1$, this gives a structure for range minima or maxima.
2. The rth largest 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 $\ell \in [2, \sigma]$ [12].
3. The same structure of the previous point can be used to find the τ-majorities in a range in time $O((1/\tau)\ell \lg n \lg_{\ell} \sigma)$ [12].
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 $O(n \lg^2 / \lg\lg n)$ bits of space [9]. Within that space, each point can be reported in time $O((\lg n/\lg\lg n)^2)$ [9]. 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 $O(r + \lg\lg n)$ [4].

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, for whichever query we want to answer. The representation must support in constant time the operations *id*, *node*, *anc*, *depth*, *leftmost*, and *lca*.
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 *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 $[1..n]$.

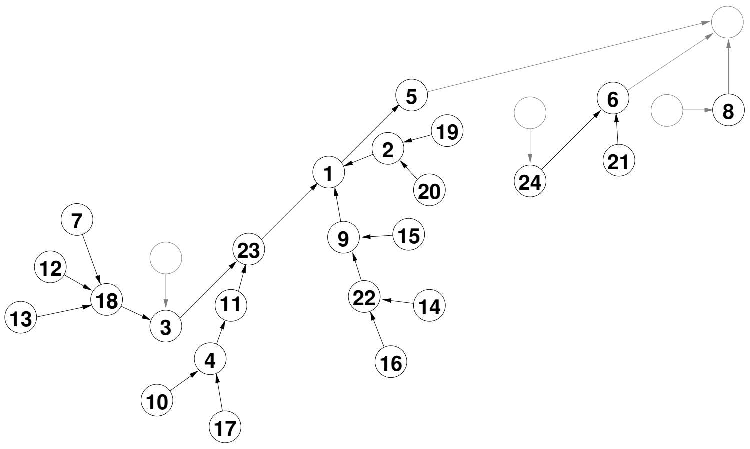

Figure 2 Our transformation to solve positive queries on functions using path queries on trees.

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 $3 \rightarrow 23 \rightarrow 1 \rightarrow 5 \rightarrow 3$, 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 $(9, 1, 5, ?)$, 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.

To handle this situation, 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).

This can be handled if the query is *decomposable*, that is, we can obtain the answer from the results on the two paths. For example, range counting and reporting are obviously decomposable, whereas range minima (if we do not store the labels, as in the solution of Section 2.3) and selection queries are not decomposable.

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 $k_2 \geq d$, we then 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 proceed as before.

Algorithm 1, in Appendix A, gives the complete procedure. We have then Theorem 1, 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 $n \lg n(1 + 1/t) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t \leq \lg n$.*

By considering the range queries of He et al. [8] (Section 2.3), we derive Corollary 2.

▶ **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 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.*

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 of v', that is, if $k_1 < d = depth(v) - depth(v')$. In this case, we set $v_s = anc(v, k_1)$. Then, if $k_2 - d < c$, we

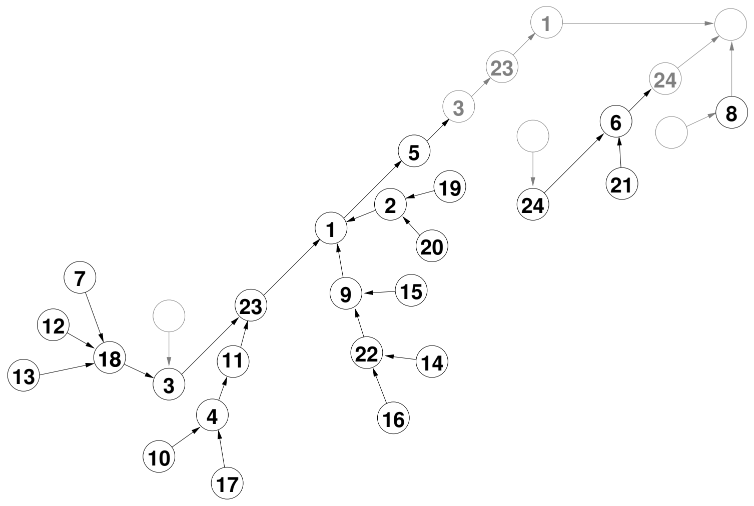

Figure 3 Our transformation to solve non-decomposable positive queries on functions using path queries on trees.

set $v_e = anc(v, k_2)$; otherwise we set $v_e = anc(v', c-1)$. Finally, we 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. Finally, if v_e is deeper than v_s, we recompute $v_e = anc(v_e, c)$. Now we can safely 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, in Appendix A, gives the pseudocode.

Since we have up to n newly created nodes for which we have to store labels, we have Theorem 3.

▶ **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 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 $n \lg n(1 + 1/t) + S(2n+1, \sigma)$ bits that answers the query in time $O(t) + T(2n+1, \sigma)$, for any $t \leq \lg n$.*

By considering the minimum/maximum and the selection queries of He et al. [8] (Section 2.3), we derive Corollaries 4 and 5.

▶ **Corollary 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. 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 $O(\lg n / \lg\lg n)$.*

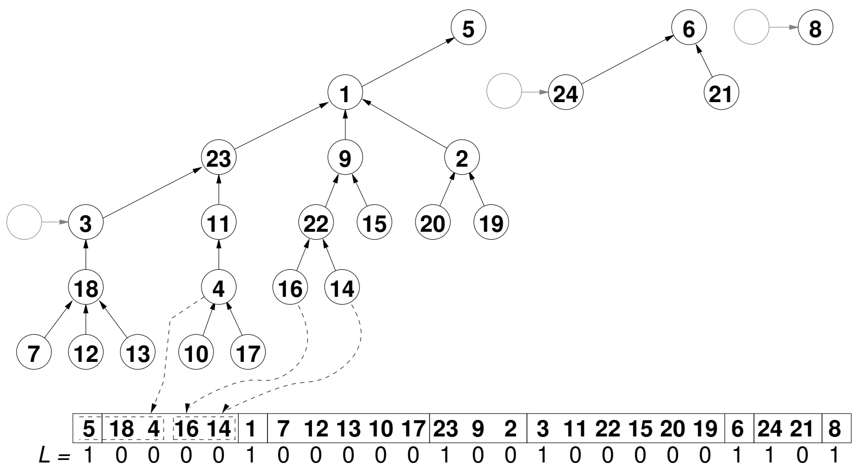

Figure 4 The same tree for decomposable positive queries (without the extra root), showing how the levels are deployed to aid in negative path points.

There exists another structure using $(1+\epsilon)n\lg n + O(m)$ bits, for any constant $\epsilon > 0$ and any $m \geq n$, that answers the queries in time $\alpha(m,n)$.

▶ **Corollary 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. 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 $H(\ell) \leq \lg\sigma$ is the entropy of the distribution of the values in ℓ.*

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 FF representation [14]. 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 $[1..n]$ 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. 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 the trees sprouting from the cycle are tall 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, $c+1$, $2c+1$, 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 $l+tc$ in each tree, for all $1 \le l \le c$. 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 $p_2 = \rho^{-1}(\mathit{rank}_1(B, \mathit{id}(v_2)))$, the second endpoint of the range, and then $p_1 = \mathit{select}_1(L, \mathit{rank}_1(L, p_2))$, 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, $v_c = \mathit{anc}(v, \mathit{depth}(v))$, $v_0 = \mathit{leftmost}(v_c)$ as the leftmost leaf,[2] and then v is in the cycle iff v_0 descends from v, that is, $v \le v_0 \le \mathit{close}(v)$.

Finally, we can build on the levelwise deployment of the node data any array range query data structure we desire. Algorithm 3, in Appendix A, 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 Theorem 6.

▶ **Theorem 6.** *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 $2n\lg n(1+1/t) + S(n,\sigma)$ bits that answers the query in time $O(t) + T(n,\sigma)$, for any $t \le \lg n$.*

By considering the various array range queries of Section 2.4, we can derive Corollaries 7 to 10, among others.

▶ **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 $2n\lg n + O(n)$ 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 $O(\lg n/\lg\lg n)$. There exists another data structure using $2n\lg n(1+1/t) + O(n)$ bits that answers the query in time $O(t)$, for any $t \le \lg n$.*

[2] This operation can be computed in this representation with $\mathit{select}_{)}(\mathit{rank}_{)}(v_c)+1)-1$ on the sequence of parentheses, which has *rank* and *select* support.

▶ **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 $2n \lg n + O(n \lg R)$ bits that finds the element with the rth largest label in $f^{-k}(i)$, for any $i \in [1..n]$, $k > 0$, and $1 \le r \le R$, in time $O(\lg n / \lg \lg n)$. It can also list the r elements with the largest or smallest values in $f^{-k}(i)$ in time $O(r + \lg n / \lg \lg n)$.*

▶ **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 $2n \lg n + (1 + \epsilon) n H(\ell) + O(n)$ 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 $O(1/\tau + \lg n / \lg \lg n)$.*

▶ **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 $2n \lg n + nH(\ell) + o(n \lg \sigma) + O(n)$ bits, 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 $O(\lg n / \lg \lg n)$. It can then list those r elements in time $O(\lg n / \lg \lg n + r(1 + \lg \sigma / \lg \lg n))$.*

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 ones. We preserve the same tree as in Section 4, but this time the mapping from nodes v is done to pairs $(\mathit{preorder}(v), \mathit{depth}(v))$. Here $\mathit{preorder}(v) = \mathit{id}(v)$ is the preorder of the node in the FF 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 $[\mathit{preorder}(v) \,..\, \mathit{preorder}(v) + \mathit{subtreesize}(v) - 1] \times [\mathit{depth}(v) + k_1 \,..\, \mathit{depth}(v) + k_2]$. All these operations are supported in constant time with the FF representation [14]. We now perform the desired query on a structure that handles two-dimensional points (possibly with labels). The returned points (p, d) are then mapped to the nodes with preorder p, $\mathit{node}(p)$, which is also supported in constant time.

For the case where v is on the cycle, we will use another arrangement. 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 $d_1 = \mathit{depth}(v) + k_1$ and $d_2 = \mathit{depth}(v) + k_2$, 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 $(\mathit{depth}(v) \text{ div } c, \mathit{depth}(v) \bmod c)$, 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 $[d_1 \bmod c \,..\, c - 1]$ and $[0 \,..\, d_2 \bmod c]$.

An exception occurs if $k_2 - k_1 \ge c$, 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 $[\mathit{preorder}(v)..\mathit{preorder}(v) + \mathit{subtreesize}(v) - 1] \times [(d_2 \text{ div } c) \cdot c \,..\, d_2]$.

Note that in this case we have to complete the query from the results of up to 3 two-dimensional ranges, so the query must be decomposable. We then obtain Theorem 11.

▶ **Theorem 11.** *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 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 $2n \lg n + O(n) + S(n, \sigma)$ 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 $2n \lg n(1 + 1/t) + S(n, \sigma)$ bits that answers the query in time $O(t) + T(n, \sigma)$, for any $t \le \lg n$.

We can combine the theorem with various results on querying two-dimensional grids of points with labels (or weights); recall Section 2.5. We obtain Corollaries 12 and 13.

▶ **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-t queries in time $O((t + \lg n) \lg^{\epsilon} n)$, 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 Conclusions

Munro et al. [10] studied how to represent an integer function $f : [1..n] \to [1..n]$ so as to efficiently find all the elements of positive and negative powers of f. We have now considered, for the first time, 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 [8, 3]. Negative powers lead to a set of domain values. For a single negative power, we basically double the space while almost retaining the performance of the corresponding array range query. For a range of negative powers, we resort to three-dimensional range queries, where time and space are essentially multiplied by $O(\lg n)$.

Our results consider queries on arbitrary labels on $[1, \sigma]$ attached to the domain elements. Appendix B gives a few improved results for the simpler case where the queries are run over the domain elements themselves.

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.

References

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

2 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 WADS*, LNCS 5664, pages 98–109, 2009.

3 T. M. Chan, M. He, J. I. Munro, and G. Zhou. Succinct indices for path minimum, with applications to path reporting. In *Proc. 22nd ESA*, LNCS 8737, pages 247–259, 2014.

4 T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th SoCG*, pages 1–10, 2011.

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

6 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.

7 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

8 M. He, J. I. Munro, and G. Zhou. Succinct data structures for path queries. In *Proc. 20th ESA*, LNCS 7501, pages 575–586, 2012.

9 J. Jájá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proc. 15th ISAAC*, pages 558–568, 2004.

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

11 G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd SODA*, pages 1066–1078, 2012.

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

13 G. Navarro, R. Raman, and S. S. Rao. Asymptotically optimal encodings for range selection. In *Proc. 34th FSTTCS*, pages 291–302, 2014.

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

15 Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

A Pseudocodes

We give detailed pseudocodes for the main procedures described in the paper. In Algorithm 1, 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.

Algorithm 1: Computing decomposable queries on positive paths.

```
1  Proc $Positive(i, k_1, k_2)$
2      $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
3      $v_c \leftarrow anc(v, depth(v) - 1)$
4      $v_1 \leftarrow anc(leftmost(v_c), 1)$
5      $c \leftarrow depth(v_1)$
6      $v' \leftarrow lca(v_1, v)$
7      $d \leftarrow depth(v) - depth(v')$
8      if $k_1 < d$ then
9          Compute path query from $anc(v, k_1)$ to $anc(v, \min(k_2, d - 1))$
10     if $k_2 \geq d$ then
11         $k'_1 \leftarrow \max(0, k_1 - d)$
12         $k'_2 \leftarrow k_2 - d$
13         if $k'_2 - k'_1 \geq c - 1$ then
14             Compute path query from $v_1$ to $v_c$
15         else
16             if $k'_1 < depth(v')$ then $v_s \leftarrow anc(v', k'_1)$
17             else $v_s \leftarrow anc(v_1, (k'_1 - depth(v')) \bmod c)$
18             if $k'_2 < depth(v')$ then $v_e \leftarrow anc(v', k'_2)$
19             else $v_e \leftarrow anc(v_1, (k'_2 - depth(v')) \bmod c)$
20             if $depth(v_s) \geq depth(v_e)$ and $anc(v_s, depth(v_s) - depth(v_e)) = v_e$ then
21                 Compute path query from $v_s$ to $v_e$
22             else
23                 Compute path query from $v_s$ to $v_c$
24                 Compute path query from $v_1$ to $v_e$
25     Return the composition of all the path queries performed; resulting nodes $u$ are
       converted into domain values $\pi(rank_1(B, id(u)))$
```

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

```
1  Proc $Positive(i, k_1, k_2)$
2      $v \leftarrow node(select_1(B, \pi^{-1}(i)))$
3      $v'_{c-1} \leftarrow anc(v, depth(v) - 1)$
4      $v_1 \leftarrow anc(leftmost(v'_{c-1}), 1)$
5      $c \leftarrow (depth(v_1) + 1)/2$
6      $v' \leftarrow lca(v_1, v)$
7      $d \leftarrow depth(v) - depth(v')$
8      if $k_1 < d$ then
9          $v_s \leftarrow anc(v, k_1)$
10         if $k_2 - d < c$ then $v_e \leftarrow anc(v, k_2)$
11         else $v_e \leftarrow anc(v', c - 1)$
12     else if $k_2 - k_1 \geq c$ then
13         $v_s \leftarrow v_1$
14         $v_e \leftarrow v_c$
15     else
16         if $depth(v) - k_1 \geq c$ then $v_s \leftarrow anc(v, k_1)$
17         else $v_s \leftarrow anc(v_1, ((c - 1) - (depth(v) - k_1)) \bmod c)$
18         if $depth(v) - k_2 \geq c$ then $v_e \leftarrow anc(v, k_2)$
19         else $v_e \leftarrow anc(v_1, ((c - 1) - (depth(v) - k_2)) \bmod c)$
20         if $depth(v_s) < depth(v_e)$ then
21             $v_e \leftarrow anc(v_e, c)$
22     Compute path query from $v_s$ to $v_e$
23     Return the answers; resulting nodes $u$ are converted into domain values
       $\pi(rank_1(B, id(u)))$, but if $depth(u) < c$ we first set $u \leftarrow anc(v_1, (c - 1) - depth(u))$
```

Algorithm 3: Computing queries on negative path points.

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

B Functions Without Labels

In the simple case where the function has no assigned labels, or said another way, we may assume $\ell(i) = i$ for the queries, we can do better than Corollaries 2 and 5. Both path query structures [8] store the sequence of labels (now domain elements) in node identifier order, and represent it with a wavelet tree [7]. 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 5, 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 14.** *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 15.** *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 $O(\lg n/\lg\lg n)$.*

We can also simplify Corollary 10, where the structures used perform the equivalent to *select* queries on the sequence of labels. Here, we can find where the domain value i appears in the sequence, and then 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.[3]

▶ **Corollary 16.** *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 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 $O((r + 1) \lg n/\lg\lg n)$.*

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