Algorithmica (2017) 78:453–491
DOI 10.1007/s00453-016-0170-7

Succinct Indices for Path Minimum, with Applications

Timothy M. Chan[1] · Meng He[2] · J. Ian Munro[1] · Gelin Zhou[1]

Received: 6 August 2015 / Accepted: 21 May 2016 / Published online: 8 June 2016
© Springer Science+Business Media New York 2016

Abstract In the *path minimum* problem, we preprocess a tree on n weighted nodes, such that given an arbitrary path, the node with the smallest weight along this path can be located. We design novel succinct indices for this problem under the *indexing model*, for which weights of nodes are read-only and can be accessed with ranks of nodes in the preorder traversal sequence of the input tree. We present

- an index within $O(m)$ bits of additional space that supports queries in $O(\alpha(m, n))$ time and $O(\alpha(m, n))$ accesses to the weights of nodes, for any integer $m \geq n$; and
- an index within $2n + o(n)$ bits of additional space that supports queries in $O(\alpha(n))$ time and $O(\alpha(n))$ accesses to the weights of nodes.

Here $\alpha(m, n)$ is the inverse-Ackermann function, and $\alpha(n) = \alpha(n, n)$. These indices give us the first succinct data structures for the path minimum problem. Following the same approach, we also develop succinct data structures for *semigroup path sum*

The partial preliminary version of this article was published in Proceedings of the 22th Annual European Symposium on Algorithms (ESA 2014) [12]. This work was supported by NSERC and the Canada Research Chairs Program. Part of the first author's work was done during his visit to the Department of Computer Science and Engineering, Hong Kong University of Science and Technology.

✉ Gelin Zhou
g5zhou@uwaterloo.ca

Timothy M. Chan
tmchan@uwaterloo.ca

Meng He
mhe@cs.dal.ca

J. Ian Munro
imunro@uwaterloo.ca

[1] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada

[2] Faculty of Computer Science, Dalhousie University, Halifax, Canada

queries, for which a query asks for the sum of weights along a given query path. One of our data structures requires $n \lg \sigma + 2n + o(n \lg \sigma)$ bits of space and $O(\boldsymbol{\alpha}(n))$ query time, where σ is the size of the semigroup. In the *path reporting* problem, queries ask for the nodes along a query path whose weights are within a two-sided query range. Using the succinct indices for path minimum queries, we achieve three different time/space tradeoffs for path reporting by designing

- an $O(n)$-word data structure with $O(\lg^{\epsilon} n + occ \cdot \lg^{\epsilon} n)$ query time;
- an $O(n \lg \lg n)$-word data structure with $O(\lg \lg n + occ \cdot \lg \lg n)$ query time; and
- an $O(n \lg^{\epsilon} n)$-word data structure with $O(\lg \lg n + occ)$ query time.

Here occ is the number of nodes reported and ϵ is an arbitrary constant between 0 and 1. These tradeoffs match the state of the art of two-dimensional orthogonal range reporting queries (Chan et al. 2011), which can be treated as a special case of path reporting queries. When the number of distinct weights is much smaller than n, we further improve both the query time and the space cost of these three results.

Keywords Path minimum · Semigroup path sum · Path reporting · Succinct data structures · Succinct encoding of directed topology trees

1 Introduction

As one of the most fundamental structures in computer science, trees generalize linear lists and have been widely used in data modeling and data representation. In many cases, objects are represented by nodes and their properties are characterized by weights assigned to nodes. Researchers have studied the problems of supporting *path queries*, that is, the schemes to preprocess a weighted tree such that various functions over the weights of nodes on a given query path can be computed efficiently [1,9,14,17,20,21,35,36,39,40,42,46].

In this article, we first consider the *path minimum (path maximum)* problem and then the more general *semigroup path sum* problem.

- *Path minimum (maximum)* Given nodes u and v, return the minimum (maximum) node along the path from u to v, i.e., the node along the path whose weight is the minimum (maximum) one;
- *Semigroup path sum* Given nodes u and v, return the sum of weights along the path from u to v, where the weights of nodes are drawn from a semigroup.

We design novel succinct data structures for these two types of path queries.

Then we revisit the problem of supporting *path reporting* queries.

- *Path reporting* Given nodes u and v along with a two-sided query range, report the nodes along the path from u to v whose weights are in the query range.

The indexing structures for path minimum queries will play a central role in our approach to path reporting queries.

When the input tree is a single path, path minimum, semigroup path sum and path reporting queries become range minimum [17,25], semigroup range sum [52] and two-dimensional orthogonal range reporting queries [13], respectively. As stated in [35],

the path queries we consider generalize these fundamental range queries to weighted trees.

In this article, we represent the input tree as an ordinal one, i.e., a rooted tree in which siblings are ordered. We use lg to denote the base-2 logarithm and use ϵ to denote a constant in $(0, 1)$. Unless otherwise specified, the underlying model of computation is the standard word RAM model with word size $w = \Omega(\lg n)$.

To present our results, we assume the following definition for the Ackermann function. For integers $\ell \geq 0$ and $i > 1$, we have

$$A_\ell(i) = \begin{cases} i+1 & \text{if } \ell = 0, \\ A_{\ell-1}^{(i+1)}(i+13) & \text{if } \ell > 0, \end{cases}$$

where $A_{\ell-1}^{(0)}(i) = i$ and $A_{\ell-1}^{(i)}(j) = A_{\ell-1}(A_{\ell-1}^{(i-1)}(j))$ for $i \geq 1$. This is growing faster than the one defined by Cormen et al. [16]. Let $\boldsymbol{\alpha}(m, n)$ be the smallest L such that $A_L(\lfloor m/n \rfloor) > n$, and $\boldsymbol{\alpha}(n)$ be $\boldsymbol{\alpha}(n, n)$. Here $\boldsymbol{\alpha}(m, n)$ and $\boldsymbol{\alpha}(n)$ are both referred to as the inverse-Ackermann functions, and are of the same order as the ones defined by Cormen et al. [16].

1.1 Path Minimum

The *minimum spanning tree verification* problem asks whether a given spanning tree is minimum with respect to a graph with weighted edges. This problem can be regarded as a special offline case of the path minimum problem, for which all the queries are processed in a single batch. Under the word RAM model [40], this problem can be solved using $O(n + m)$ comparisons and linear overhead, where n and m are the numbers of nodes and edges, respectively. See [11,15,19,41] for other results under different models. The online path minimum problem requires slightly more comparisons. As shown by Pettie [47], $\Omega(q \cdot \boldsymbol{\alpha}(q, n) + n)$ comparisons are necessary to serve q queries over a tree of size n.

Data structures for the path minimum problem have been heavily studied. An early result presented by Alon and Schieber [1] requires $O(n)$ words of space and $O(\boldsymbol{\alpha}(n))$ query time. Since then, several solutions using $O(n)$ words, i.e., $O(n \lg n)$ bits, with constant query time have been designed under the word RAM model [3,9,14,17,39]. Chazelle [14] and Demaine et al. [17] generalized Cartesian trees [51] to weighted trees and used them to support path minimum queries. Alstrup and Holm [3] and Brodal et al. [9] made use of macro-micro decomposition in designing their data structures. The solution of Kaplan and Shafrir [39] is based on Gabow's recursive decomposition of trees [29].

In this article we present lower and upper bounds for path minimum queries. In Lemma 3.1 we show that $\Omega(n \lg n)$ bits of space are necessary to encode the answers to path minimum queries over a tree of size n. This distinguishes path minimum queries from range minimum queries in terms of space cost, for which $2n$ bits are sufficient to encode all answers over an array of size n [25].

We adopt the *indexing model* (also called the *systematic model*) [4,7,10] in designing new data structures for path minimum queries. Applying this model to weighted

trees, we assume that weights of nodes are represented in an arbitrary given form; the only requirement is that the representation supports access to the weight of a node given its preorder rank, i.e., the rank of the node in the preorder traversal sequence of the weighted tree. Auxiliary data structures called *indices* are then constructed, and query algorithms use indices and the access operator provided for the raw data. This model is theoretically important and its variants are frequently used to prove lower bounds [18,31,43]. In addition, the indexing model is also of practical importance as it addresses cases in which the (large) raw data are stored in slower external memory or even remotely, while the (smaller) indices could be stored in memory or locally. The space of an index is called *additional space*. Note that the lower bound in the previous paragraph is proved under the encoding model, and thus does not apply to the indexing model.

The following theorem presents our indices for path minimum.

Theorem 1.1 *An ordinal tree on n weighted nodes can be indexed (a) using $O(m)$ bits of space and $O(m)$ construction time to support path minimum queries in $O(\boldsymbol{\alpha}(m,n))$ time and $O(\boldsymbol{\alpha}(m,n))$ accesses to the weights of nodes, for any integer $m \geq n$; or (b) using $2n + o(n)$ bits of space and $O(n)$ construction time to support path minimum queries in $O(\boldsymbol{\alpha}(n))$ time and $O(\boldsymbol{\alpha}(n))$ accesses to the weights of nodes.*

To better understand variant (a) of this result, we discuss the time and space costs for the following possible values of m. When $m = n$, then we have an index of $O(n)$ bits that supports path minimum queries in $O(\boldsymbol{\alpha}(n))$ time. When $m = \Theta(n(\lg^*)^* n)$, for example, then it is well-known that $\boldsymbol{\alpha}(m,n) = O(1)$, and thus we have an index of $O(n(\lg^*)^* n)$ bits that supports path minimum queries in $O(1)$ time[1]. Combining the above index with a trivial encoding of node weights, we obtain data structures for path minimum queries with $O(1)$ query time and almost linear bits of additional space. Previous solutions [3,9,14,17,39] to the same problem with constant query time occupy $\Omega(n \lg n)$ bits of space in addition to the space required for the input tree.

Taking the construction time into account, variant (a) with $m = \max\{q, n\}$ gives us a data structure that answers q path minimum queries in $O(q \cdot \boldsymbol{\alpha}(q,n) + \max\{q,n\}) = O(q \cdot \boldsymbol{\alpha}(q,n) + n)$ time, which matches the lower bound of Pettie [47].

Finally, variant (b) gives us the first succinct data structure for path minimum queries, which occupies an amount of space that is close to the information-theoretic lower bound of storing a weighted tree. With a little extra work, we can even represent a weighted tree using $n \lg \sigma + 2n + o(n)$ bits only, i.e., within $o(n)$ additive term of the information-theoretic lower bound, to support queries in $O(\boldsymbol{\alpha}(n))$ time.

1.2 Semigroup Path Sum

Generalizing the semigroup range sum queries on linear lists [52], the problem of supporting semigroup path sum queries has been considered by Alon and Schieber [1] and Chazelle [14]. Alon and Schieber designed a data structure with $O(n)$ words of

[1] The function $(\lg^*)^*$ is the number of times $\lg^*$ must be iteratively applied before the result becomes less than or equal to 1. See Nivasch's discussions [45] for more details.

space and construction time that supports semigroup path sum queries in $O(\boldsymbol{\alpha}(n))$ time. Unlike Alon and Schieber's and our formulation, Chazelle considered trees on weighted edges instead of weighted nodes. However, it is not hard to see that these two formulations are equivalent. Chazelle further showed that, for any $m \geq n$, one could obtain a word-RAM data structure with $O(\boldsymbol{\alpha}(m, n))$ query time in addition to $O(m)$ construction time and words of space. The solution of Chazelle is optimal, as established in the lower bound of Yao [52].

Our data structures for semigroup path sum queries are summarized in the following theorem.

Theorem 1.2 *Let T be an ordinal tree on n nodes, each having a weight drawn from a semigroup of size σ. Then T can be stored (a) using $m \lg \sigma + 2n + o(n)$ bits of space and $O(m)$ construction time to support semigroup path sum queries in $O(\boldsymbol{\alpha}(m, n))$ time, for some constant $c > 1$ and any integer $m \geq cn$; or (b) using $n \lg \sigma + 2n + o(n \lg \sigma)$ bits of space and $O(n)$ construction time to support semigroup path sum queries in $O(\boldsymbol{\alpha}(n))$ time.*

Variant (a) matches the data structures of Chazelle [14], and our approach can be further used to achieve variant (b), which is the first succinct data structure with near-constant query time for the semigroup path sum problem. Since path minimum queries are special cases of semigroup path sum queries, the data structures described in Theorem 1.2 can be directly used for path minimum queries at no extra cost. However, these structures cannot achieve both linear space and constant query time.

1.3 Path Reporting

Path reporting queries were proposed by He et al. [35]. They obtained two solutions: one uses $O(n)$ words and $O(\lg \sigma + occ \cdot \lg \sigma)$ query time, and the other uses $O(n \lg \lg \sigma)$ words but $O(\lg \sigma + occ \cdot \lg \lg \sigma)$ query time, where σ is the number of distinct weights and occ is the number of nodes reported. For the same problem, Patil et al. [46] designed a succinct structure based on *heavy path decomposition* [33,50]. Their structure requires only $n \lg \sigma + 6n + o(n \lg \sigma)$ bits but $O(\lg \sigma \lg n + occ \cdot \lg \sigma)$ query time. Concurrently, He et al. [36] designed another succinct structure based on a different idea. This structure, requiring $O((\lg \sigma / \lg \lg n + 1) \cdot (1 + occ))$ query time and $nH(W_T) + 2n + o(n \lg \sigma)$ bits of space, where $H(W_T)$ is the entropy of the multiset of the weights of the nodes in the input tree T, is the best previously known linear space solution.

In this article, we design three new data structures for path reporting queries:

Theorem 1.3 *An ordinal tree on n nodes whose weights are drawn from a set of σ distinct weights can be represented using $O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits of space, so that path reporting queries can be supported in $O(\min\{\lg \lg \sigma + \mathtt{t}(\sigma), \lg \sigma / \lg \lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg \sigma / \lg \lg n + 1\})$ time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^{\epsilon} \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg \lg \sigma)$ and $\mathtt{t}(\sigma) = O(\lg \lg \sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^{\epsilon} \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

Table 1 Our data structures for path reporting queries, along with previous results on path reporting and two-dimensional orthogonal range reporting (which are marked by †)

Source	Space	Query time
He et al. [35]	$O(n)$ words	$O(\lg\sigma \cdot (1+occ))$
He et al. [35]	$O(n \lg\lg\sigma)$ words	$O(\lg\sigma + occ \lg\lg\sigma)$
Patil et al. [46]	$n\lg\sigma + 6n + o(n\lg\sigma)$ bits	$O(\lg\sigma \lg n + occ \lg\sigma)$
He et al. [36]	$nH(W_T) + 2n + o(n\lg\sigma)$ bits	$O((\lg\sigma/\lg\lg n + 1)\cdot(1+occ))$
Bose et al. [6]†	$n\lg\sigma + o(n\lg\sigma)$ bits	$O((\lg\sigma/\lg\lg n + 1)\cdot(1+occ))$
Chan et al. [13]†	$O(n)$ words	$O(\lg\lg n + \lg^{\epsilon}\sigma + occ\lg^{\epsilon}\sigma)$
Chan et al. [13]†	$O(n\lg\lg\sigma)$ words	$O(\lg\lg n + occ\lg\lg\sigma)$
Chan et al. [13]†	$O(n\lg^{\epsilon}\sigma)$ words	$O(\lg\lg n + occ)$
New	$O(n\lg\sigma)$ bits	$O(\min\{\lg^{\epsilon}\sigma, \lg\sigma/\lg\lg n + 1\}\cdot(1+occ))$
New	$O(n\lg\sigma\lg\lg\sigma)$ bits	$O(\min\{\lg\lg\sigma, \lg\sigma/\lg\lg n + 1\}\cdot(1+occ))$
New	$O(n\lg^{1+\epsilon}\sigma)$ bits	$O(\min\{\lg\lg\sigma, \lg\sigma/\lg\lg n + 1\} + occ)$

All of these results assume the standard word RAM model with word size $w = \Omega(\lg n)$. Here $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T. Note that $H(W_T)$ is at most $\lg\sigma$, which is $O(w)$

These results completely subsume almost all previous results; the only exceptions are the succinct data structures for this problem designed in previous work, whose query times are worse than our linear-space solution. Furthermore, our data structures match the state of the art of 2D range reporting queries [13] when $\sigma = n$, and have better performance when σ is much less than n. We compare our results with previous work on path reporting in Table 1.

1.4 An Overview of the Article

The rest of this article is organized as follows. Section 2 reviews previous techniques and terminology that we will use for our data structures.

In Sects. 3 and 4, we design novel succinct data structures for the path minimum problem and the semigroup path sum problem. Unlike previous succinct tree structures [22,30,34,36,46], our approach is based on Frederickson's *restricted topological partitions* [28], which transform the input tree into a binary tree and further recursively decompose it into a hierarchy of clusters with constant external degrees and logarithmically many levels. The hierarchy is referred to as a *directed topology tree*. Our main strategy of constructing query-answering structures is to recursively divide the set of levels of hierarchy into multiple subsets of levels; with a carefully-defined variant of the query problem which takes levels in the hierarchy as parameters, the query over the entire structure can be answered by conquering the subproblems local to the subsets of levels. Solutions to special cases of the query problem are also designed, so that we can present the time and space costs of our solution using recursive formulas. Then, by carefully constructing number series and using them in the division of levels into subsets, we can prove that our structures achieve the tradeoff presented in Theorems 1.1

to 1.2 using the inverse-Ackermann function. This approach is novel and exciting in the design of succinct data structures, and it does not directly use standard techniques for word RAM at all.

The above strategy would not achieve the desired space bound without a succinct data structure that supports navigation in the input tree, the binary tree that it is transformed into and the clusters in the directed topology tree. In Sect. 5, we design such a structure occupying only $2n + o(n)$ bits, which is of independent interest.

In Sects. 6 and 7, to design solutions to path reporting, we follow the general framework of He et al. [36] to extract subtrees based on the partitions of the entire weight range, and make use of a conceptual structure that borrows ideas from the classical range tree. One strategy of achieving improved results is to further reduce path reporting into queries in which the weight ranges are one-sided, which allows us to apply our succinct index for path minimum queries to achieve the tradeoffs presented in the second half of the abstract. We further apply a tree covering strategy to reduce the space cost for the case in which the number of distinct weights is much smaller than n, and hence prove Theorem 1.3.

Finally, we end this article with some open problems in Sect. 8.

2 Preliminaries

2.1 Restricted Topological Partitions and Directed Topology Trees

Topological partitions and restricted topological partitions have found applications in computing the k smallest spanning trees of a graph [26,27], and in dynamic maintenance of minimum spanning trees and connectivity information [26], 2-edge-connectivity information [27], and a set of rooted trees that support link-cut operations [28]. In this article, we follow the definitions and notation of restricted topological partitions and directed topology trees [28].

Let $\mathcal{B}$ be a rooted binary tree. A cluster with respect to $\mathcal{B}$ is a subset of nodes whose induced subgraph forms a connected component. The *external degree* of a cluster is the number of edges that have exactly one endpoint in the cluster. These endpoints are referred to as the *boundary nodes* of the cluster. For two disjoint clusters C_1 and C_2, C_1 is said to be a *child cluster* of C_2 if C_1 contains a node whose parent is contained in C_2.

The binary tree $\mathcal{B}$ is then partitioned into a hierarchy of clusters as follows.

Lemma 2.1 ([28]). *A binary tree $\mathcal{B}$ on n nodes can be partitioned into a hierarchy of clusters with $h + 1$ levels for some $h = O(\lg n)$, such that,*

- *the clusters at level 0 each contain a single node, and the only cluster at level h contains all the nodes of $\mathcal{B}$;*
- *for each level $i > 0$, each cluster at level i is the disjoint union of at most 2 clusters at level $i - 1$;*
- *for each level $i = 0, 1, 2, \ldots, h$, there are at most $(5/6)^i n$ clusters of sizes at most 2^i, which form a partition of the nodes in the binary tree;*
- *each cluster is of external degree at most 3 and contains at most two boundary nodes; and*

- *any cluster that has more than one child cluster contains only a single node.*

The hierarchy of clusters is referred to as the directed topology tree of $\mathcal{B}$, which is denoted by $\mathcal{D}$. A node at level i, where $i > 0$, of $\mathcal{D}$ represents a cluster C at level i of the hierarchy, and its children represent the clusters at the lower level that partition C. In particular, the leaf nodes of $\mathcal{D}$, which are at level 0, represent individual nodes of the binary tree $\mathcal{B}$. We illustrate these concepts in Fig. 3.

2.2 Tree Extraction

To support path queries, He et al. [35,36] presented the technique of tree extraction. This technique is based on the deletion operation of tree edit distance [5]. To delete a non-root node u, its children are inserted in place of u into the list of children of its parent, preserving the original left-to-right order. Let T be an ordinal tree and X be a subset of nodes in T. The X-extraction of T, F_X, is defined to be the ordinal forest obtained by deleting all the nodes that are not in X from T. There is a natural one-to-one correspondence between the nodes in X and the nodes in F_X, and the ancestor-descendant and preorder relationships among the remaining nodes are preserved. If X contains the root of T, then F_X consists of a single ordinal tree only, which is denoted by T_X.

He et al. [35,36] further defined notation in terms of weights. Let T be an ordinal tree on nodes whose weights are drawn from $[1..\sigma]$. For any range $[a..b] \subseteq [1..\sigma]$, they defined $R_{a,b}$ to be the set of nodes in T whose weights are in $[a..b]$. They also defined $anc_{a,b}(T, x)$ to be the lowest ancestor of x whose weight is in $[a..b]$; $anc_{a,b}(T, x)$ is defined to be `dummy` if no such ancestor exists. In addition, they defined $F_{a,b}$ to be the ordinal forest obtained by deleting from T all the nodes that are not in $R_{a,b}$, where the nodes are deleted from bottom to top. Note that there is a one-to-one mapping between the nodes in $R_{a,b}$ and the nodes in $F_{a,b}$. As proved in [35,36], the nodes in $R_{a,b}$ and the nodes in $F_{a,b}$ that correspond to them have the same relative positions in the preorder traversal sequences of T and $F_{a,b}$.

2.3 Bit Vectors and Sequences

Bit vectors are one of the main building blocks in many space efficient data structures. Let $B[1..n]$ denote a bit vector of size n. For $\alpha \in \{0, 1\}$, $\texttt{rank}_\alpha(B, i)$ counts α-bits in $B[1..i]$, while $\texttt{select}_\alpha(B, i)$ finds the i-th α-bit in B. The problem of representing bit vectors succinctly is addressed in the following lemma.

Lemma 2.2 ([48]). *A bit vector with $n-m$ zeros and m ones can be represented using $\lg \binom{n}{m} + O(n \lg\lg n / \lg n)$ bits of space to support $\texttt{rank}_\alpha$, $\texttt{select}_\alpha$, and the access to each bit in $O(1)$ time.*

Bit vectors can be generalized to sequences of labels that are drawn from an alphabet Σ of size σ. Operations $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ are also generalized by setting $\alpha \in \Sigma$. For sequences with $\sigma = O(\lg^\epsilon n)$, Ferragina et al. [24] designed a succinct representation to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in $O(1)$ time. Using this succinct

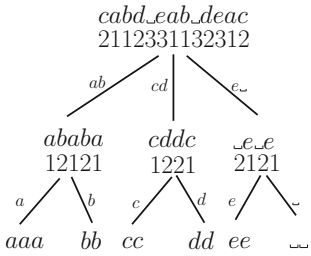

Fig. 1 An example of generalized wavelet trees in which $S = cabd\_eab\_deac$ and $\Sigma = \{a, b, c, d, e, \_\}$. We set $f = 3$ and list $\widetilde{S}_v$ sequences on internal nodes

representation as a building block, the same authors presented generalized wavelet trees [24] to encode sequences over general alphabets.

A generalized wavelet tree for a sequence $S[1..n]$ over an alphabet Σ is constructed by recursively splitting the alphabet into $f = \lceil \lg^{\epsilon} n \rceil$ subsets of almost equal sizes. Each node in that tree is associated with a subset of labels Σ_v, and a subsequence S_v of S that consists of the positions whose labels are in Σ_v. In particular, each leaf is associated with a single label.

At each non-leaf node v, a sequence $\widetilde{S}_v$ of labels drawn from $[1..f]$ is created according to S_v. Formally, suppose that the children of v are $v_1, v_2, \ldots, v_f$, $\widetilde{S}_v[i] = \alpha$ if and only if $S_v[i] \in \Sigma_{v_\alpha}$. $\widetilde{S}_v$ is stored using the succinct representation described above, such that $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations can be supported in constant time. See Fig. 1 for an example.

Each position in S_v corresponds to a position in S. Chan et al. [13] studied the following *ball-inheritance* problem: given an arbitrary position in some S_v, find the corresponding position in S and the label at this position. Their solution is summarized in Lemma 2.3. Note that the original solution was developed for binary wavelet trees. However, their approach can be directly extended to generalized wavelet trees.

Lemma 2.3 ([13]). *Let $S[1..n]$ be a sequence of labels that are drawn from $[1..\sigma]$. Given a generalized wavelet tree of S, one can build auxiliary data structures for the ball-inheritance problem with $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of space and $O(\mathtt{t}(\sigma))$ query time, where (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^{\epsilon} \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg \lg \sigma)$ and $\mathtt{t}(\sigma) = O(\lg \lg \sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^{\epsilon} \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

2.4 Succinct Ordinal Trees Based on Tree Covering

The technique of tree covering is employed to represent an ordinal tree succinctly [22, 23,30,34]. We summarize the algorithm of Farzan and Munro [22] for computing tree covering in the following lemma.

Lemma 2.4 ([22, Theorem 1] and [23, Lemma 2]). *Let T be an ordinal tree on n nodes. For a fixed parameter M, one can cover the nodes in T by $\Theta(n/M)$ cover elements (i.e., subtrees) of size up to $2M$, all of which are pairwise disjoint other than their root nodes. In addition, there is at most one non-root node in each cover element that has a child in another cover element. Consequently, nodes in one cover element are distributed into $O(1)$ preorder segments, i.e., maximal substrings of nodes in the preorder traversal sequence of T that are in the same cover element.*

Based on tree covering, researchers [22,30,34] designed succinct representations for ordinal trees over an alphabet of size $\sigma = o(\lg \lg n)$. Unlabeled trees can be regarded as a special case in which $\sigma = 1$. Their results are summarized in Lemma 2.5.

Lemma 2.5 ([22,30,34]). *An ordinal tree T on n nodes over an alphabet of size $\sigma = o(\lg \lg n)$ can be encoded in $n(\lg \sigma + 2) + O(\sigma n \lg \lg \lg n / \lg \lg n)$ bits of space to support the following operations in $O(1)$ time. Here x and y, which are nodes in T, are identified by preorder ranks. A node is its own 0-th ancestor. In addition, a node with label α is an α-node, and an α-node is an α-ancestor of its descendants.*

- $\texttt{depth}(T, x)$*: the depth of x (i.e., the number of ancestors of x);*
- $\texttt{depth}_\alpha(T, x)$*: the number of α-ancestors of x;*
- $\texttt{parent}(T, x)$*: the parent of x;*
- $\texttt{level\_anc}(T, x, i)$*: the i-th lowest ancestor of x;*
- $\texttt{level\_anc}_\alpha(T, x, i)$*: the i-th lowest α-ancestor of x;*
- $\texttt{LCA}(T, x, y)$*: the lowest common ancestor of x and y.*

This data structure can be constructed in $O(n)$ time.

3 Path Minimum Queries

3.1 A Lower Bound Under the Encoding Model

We first give a simple lower bound for path minimum queries under the *encoding model*, i.e., the least number of bits required to encode the answers to all possible queries.

Lemma 3.1 *In the worst case, $\Omega(n \lg n)$ bits are required to encode the answers to all possible path minimum queries over a tree on n weighted nodes.*

Proof Consider a tree T with $n_L = \Theta(n)$ leaves. We assign the smallest n_L distinct weights to these leaves, and assign larger weights to the other nodes. It follows that the smallest weight on any path from a leaf to another must appear at one of its endpoints. The order of the weights assigned to leaf nodes, which requires $\lg(n_L!) = \Omega(n \lg n)$ bits to encode, can be fully recovered using path minimum queries. Therefore, $\Omega(n \lg n)$ bits are necessary to encode the answers to path minimum queries over T. □

While the lower bound of Pettie [47] focuses on the overall processing time, Lemma 3.1 provides a separation between path minimum and range minimum in terms of space: $\Omega(n \lg n)$ bits are required to encode path minimum queries over a tree on n weighted nodes, while range minimum over an array of length n can always be encoded in $2n$ bits [25].

3.2 Upper Bounds Under the Indexing Model

Now we consider the support for path minimum queries. The space cost of maintaining a weighted tree is dominated by storing the weights of nodes. Thus we represent the

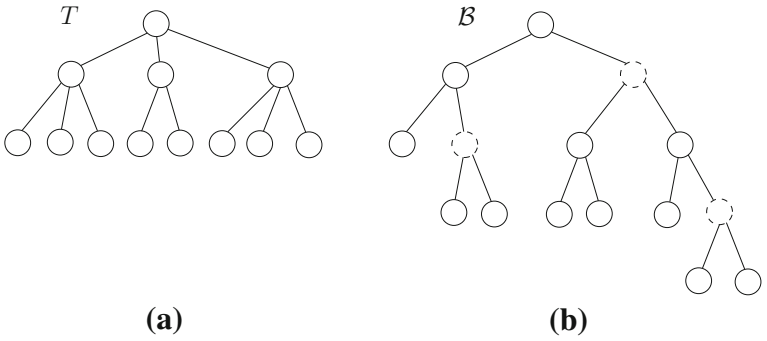

Fig. 2 An illustration of the binary tree transformation. **a** An input tree T on 12 nodes. **b** The transformed binary tree $\mathcal{B}$, where dummy nodes are represented by *dashed circles*

input tree as an ordinal one, for which the nodes are identified by their preorder ranks. This strategy has no significant impact to the space cost. We will assume the indexing model described in Sect. 1 and develop several novel succinct indices for path minimum queries. In these data structures, the weights of nodes are assumed to be stored separately from the index for queries, and can be accessed with the preorder ranks of nodes. The time cost to answer a given query is measured by the number of accesses to the index and that to node weights.

Let T be an input tree on n nodes. Here T is represented as an ordinal one, and its nodes are identified by preorder ranks. As illustrated in Fig. 2, we transform T into a binary tree, $\mathcal{B}$, of size at most $2n$ as follows (essentially as in the usual way but with added dummy nodes): For each node u with $d > 2$ children, where $v_1, v_2, \ldots, v_d$ are children of u, we add $d-2$ dummy nodes $x_1, x_2, \ldots, x_{d-2}$. The left and right children of u are set to be v_1 and x_1, respectively. For $1 \leq k < d-2$, the left and right children of x_k are set to be v_{k+1} and x_{k+1}, respectively. Finally, the left and right children of x_{d-2} are set to be v_{d-1} and v_d, respectively. In this way we have replaced u and its children with a right-leaning binary tree, where the leaf nodes are children of u. This transformation does not change the preorder relationship among the nodes in T. In addition, the set of non-dummy nodes along the path between any two non-dummy nodes remain the same after transformation.

As illustrated in Fig. 3, we decompose $\mathcal{B}$ and obtain the directed topology tree $\mathcal{D}$ using Lemma 2.1. As T and $\mathcal{B}$ are rooted trees, each cluster contains a node that is the ancestor of all the other nodes in the same cluster. This node is referred to as the *head* of the cluster. Note that the head of a cluster is also a boundary node except for the cluster that includes the root of $\mathcal{B}$. For a cluster that has two boundary nodes, the non-head one is referred to as the *tail* of the cluster. If the head and the tail of a cluster are not adjacent, then the path between but excluding them is said to be the *spine* of the cluster, i.e., the spine is obtained by removing the head and the tail from the path that connects them.

In the directed topology tree $\mathcal{D}$, sibling clusters are ordered by the preorder ranks of their heads. Each cluster C is identified by its *topological rank*, i.e., the preorder rank of the node in $\mathcal{D}$ that represents C. For simplicity, a cluster at level i is called a *level-i cluster*, and its boundary nodes are said to be *level-i boundary nodes*. To facilitate the

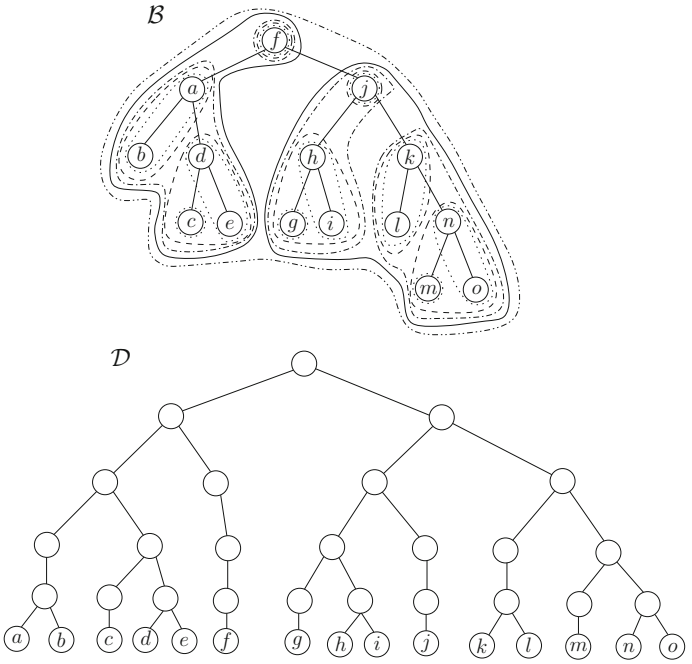

Fig. 3 The multilevel restricted topological partitions and the directed topology tree $\mathcal{D}$ for the binary tree $\mathcal{B}$ shown in Fig. 2b

use of directed topology trees, we define the following operations relevant to nodes, clusters, boundary nodes and spines. The support for these operations is summarized in the following lemma.

Lemma 3.2 *Let T be an ordinal tree on n nodes. Then T, the transformed binary tree $\mathcal{B}$, and their directed topology tree $\mathcal{D}$ can be encoded using $O(n)$ construction time and $2n + o(n)$ bits of space, such that the following operations can be supported in $O(1)$ query time. Here x and y are nodes in $\mathcal{B}$, and C is a cluster in $\mathcal{D}$.*

- *conversions between nodes in T and $\mathcal{B}$;*
- `level_cluster`$(\mathcal{D}, i, x)$*: return the level-i cluster that contains node x;*
- `LLC`$(\mathcal{D}, x, y)$*: return the cluster at the lowest level that contains nodes x and y;*
- `cluster_head`$(\mathcal{D}, C)$*: return the head of cluster C;*
- `cluster_tail`$(\mathcal{D}, C)$*: return the tail of cluster C or* `NULL` *if it does not exist;*
- `cluster_spine`$(\mathcal{D}, C)$*: return the endpoints of the spine of cluster C or* `NULL` *if the spine does not exist;*
- `cluster_nn`$(\mathcal{D}, C, x)$*: return the boundary node of C that is the closest to node x, given that x is outside of C;*
- `parent`$(\mathcal{B}, x)$*: return the parent node of x in $\mathcal{B}$;*
- `LCA`$(\mathcal{B}, x, y)$*: return the lowest common ancestor of x and y;*
- `BN_rank`$(\mathcal{B}, i, x)$*: count the level-i boundary nodes that precede x in preorder of $\mathcal{B}$;*

- `BN_select`$(\mathcal{B}, i, j)$*: return the* j*-th level-*i *boundary node in preorder of* $\mathcal{B}$.

Next we describe our data structures for path minimum queries. To highlight our key strategy, we defer the proof of Lemma 3.2 to Sect. 5. As the conversion between nodes in T and $\mathcal{B}$ can be performed in $O(1)$ time, we assume that the endpoints of query paths and the minimum nodes are both specified by nodes in $\mathcal{B}$. We first consider how to find the minimum node for specific subsets of paths in $\mathcal{B}$. Let h be the highest level of $\mathcal{D}$. The following subproblems are defined in terms of clusters and boundary nodes, for $0 \le i < j \le h$.

- $\mathcal{PM}_{i,j}$: find minimum nodes along query paths between two level-i boundary nodes that are contained in the same level-j cluster;
- $\mathcal{PM}'_{i,j}$: find minimum nodes along query paths from a level-i boundary node to a level-j one (which is also a level-i boundary node), where both boundary nodes are contained in the same level-j cluster.

Thus the original problem is $\mathcal{PM}_{0,h}$. If $\mathcal{PM}_{i,j}$ is solved, then $\mathcal{PM}'_{i,j}$ and $\mathcal{PM}_{i',j}$ for $i' > i$ are also naturally solved.

We will select a set of *canonical* paths in $\mathcal{B}$, for which the minimum nodes on all these canonical paths have been precomputed and stored. By the indexing model we adopt, singleton paths are naturally canonical. In our query algorithm, each query path will always be *partitioned* into a set of *canonical subpaths*, so that each node on the query path is contained in exactly one of these canonical subpaths, and the endpoints of the query path are contained in singleton canonical paths. Let $u \sim v$ denote the path from u to v. If node t is on $u \sim v$, then the partition of $u \sim v$ could be obtained by taking the union of the partitions of $u \sim t$ and $v \sim t$, which both contain t in a singleton canonical path.

Let $h_\tau > 0$ be a parameter whose value will be determined later. For each cluster whose level is higher than or equal to h_τ, we explicitly store the minimum node on its spine, i.e., the spine is made canonical. The following lemma addresses the cost incurred.

Lemma 3.3 *It requires* $O(h_\tau(5/6)^{h_\tau} n)$ *bits of additional space and* $O(n)$ *construction time to make these spines canonical.*

Proof It requires i bits to store the minimum node on the spine of a level-i cluster, as the cluster contains at most 2^i nodes. As $\mathcal{B}$ has at most $2n$ nodes, there are at most $(5/6)^i \cdot 2n$ level-i clusters. Thus the overall space cost is $\sum_{i=h_\tau}^{h} (i(5/6)^i \cdot 2n) = O(h_\tau(5/6)^{h_\tau} n)$ bits.

The minimum nodes on the spines of level-h_τ clusters can be simply found in $O(n)$ overall time using brute-force search. For $h_\tau < i \le h$, the spine of a level-i cluster C, can be partitioned into singleton paths and spines of level-$(i-1)$ clusters that are contained in C. This requires only $O(1)$ time per cluster, as C is the disjoint union of at most 2 level-$(i-1)$ clusters. Thus the overall construction time is $O(n) + \sum_{i=h_\tau+1}^{h} ((5/6)^i \cdot 2n \cdot O(1)) = O(n)$. □

In particular, when $h_\tau = \omega(1)$, the space cost in Lemma 3.3 is $o(n)$ bits.

We will solve $\mathcal{PM}_{0,h_\tau}$ using brute-force search, and support $\mathcal{PM}_{h_\tau,h}$ using a novel recursive approach as described below. The base cases of recursion are summarized in Lemmas 3.4 to 3.6.

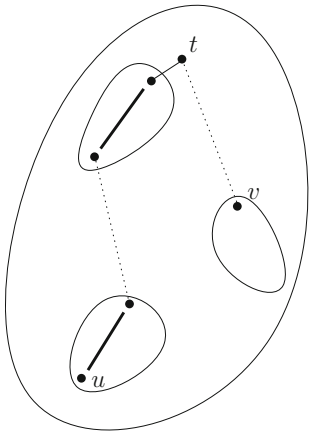

Fig. 4 An illustration for the proof of Lemma 3.5. Here the *large splinegon* represents a level-j cluster and the *small* ones represent level-i clusters contained in the level-j cluster. *Bold lines* represent spines of level-i clusters and *dotted lines* represent paths

Lemma 3.4 *$\mathcal{PM}_{0,h_\tau}$ can be solved using $O(2^{h_\tau})$ query time and no extra space.*

Proof By Lemma 2.1, each level-h_τ cluster contains at most 2^{h_τ} nodes. Thus any path of $\mathcal{PM}_{0,h_\tau}$ can be traversed within $O(2^{h_\tau})$ time using `parent` and `LCA` operations. The minimum node on the path can be found in the meanwhile. □

Lemma 3.5 *For every pair of i and j satisfying $h_\tau \le i < j \le h$ and $j - i = O(1)$, $\mathcal{PM}_{i,j}$ can be solved using $O(1)$ query time and no extra space.*

Proof Let u and v be the endpoints of some given query path of $\mathcal{PM}_{i,j}$. That is, u and v are two level-i boundary nodes that are contained in the same level-j cluster. To partition the path $u \sim v$, we first compute $t = \texttt{LCA}(\mathcal{B}, u, v)$. Node t must also be a level-i boundary node; otherwise the cluster that contains t would have at least two child clusters. As shown in Fig. 4, we then partition the path $u \sim t$ into a constant number of singleton paths and spines of level-i clusters, which are all canonical. Initially, we set $x = u$ and let C be the level-i cluster that contains x. The following procedure is repeated until x becomes the parent of t. We make use of $\texttt{cluster\_spine}(\mathcal{D}, C)$ to check whether x is on the spine of C. If x is on the spine, then y is set to be the other endpoint of the spine; otherwise $y = x$. In both cases, we select the path $x \sim y$, which must be canonical, and reset $x = \texttt{parent}(\mathcal{B}, y)$ and $C = \texttt{level\_cluster}(\mathcal{D}, i, x)$.

By Lemma 2.1, each level-j cluster is a disjoint union of a constant number of level-i clusters, as $2^{j-i} = 2^{O(1)}$ is a constant. Therefore, the path $u \sim t$ can be partitioned into $O(1)$ canonical subpaths using the procedure described above. The path $v \sim t$ can be partitioned similarly. Taking the union of these two sets of selected canonical paths except for a singleton path that contains t, we determine $O(1)$ canonical paths that the path $u \sim v$ is partitioned into, and thus the minimum node on $u \sim v$. Clearly the algorithm uses only $O(1)$ time. □

Lemma 3.6 *For a fixed pair of i and j satisfying $h_\tau \le i < j \le h$, $\mathcal{PM}'_{i,j}$ can be solved using $O(1)$ query time, with an auxiliary data structure requiring $O((5/6)^i n)$ bits of extra space and construction time.*

Proof In this proof, we will implicitly make each query path of $\mathcal{PM}'_{i,j}$ canonical and store the minimum nodes on these paths in a highly efficient way.

We construct an auxiliary ordinal tree, $T_{i,j}$, using the technique of tree extraction. The structure of $T_{i,j}$ is obtained by extracting all level-i boundary nodes from $\mathcal{B}$. By Lemma 2.1, $T_{i,j}$ consists of $O((5/6)^i n)$ nodes. For convenience, we refer to a node in $T_{i,j}$ as u' iff it corresponds to a level-i boundary node u in $\mathcal{B}$. The conversion between u and u' can be performed in $O(1)$ time using `BN_rank` and `BN_select`.

Next we assign labels from alphabet $\{0, 1\}$ to the nodes of $T_{i,j}$. We only consider the case in which the level-j boundary node is the head of its cluster; the other case can be handled similarly. Let u be any level-i boundary node and let v be the head of $C_0 = \texttt{level\_cluster}(\mathcal{B}, j, u)$, i.e., the level-$j$ cluster that contains u. As in the proof of Lemma 3.5, the path from u to v in $\mathcal{B}$ can be partitioned into a sequence of singleton paths and spines of level-i clusters. Let x' be the next node on the path from u' to v'. We assign 1 to u' in $T_{i,j}$ if $u = v$, or the minimum node on $u \sim v$ is smaller than that on $x \sim v$; otherwise we assign 0 to u'. See Fig. 5 for an example. We represent this labeled tree within $O((5/6)^i n)$ bits of space and $O((5/6)^i n)$ construction time using Lemma 2.5.

To find the minimum node between u and v, we need only to find the closest 1-node to u' along the path from u' to v' in $T_{i,j}$. This node can be found in $O(1)$ time by performing $\texttt{level\_anc}_\alpha$ and $\texttt{depth}_\alpha$ operations on $T_{i,j}$. Let x' be the node found. Then the minimum node on $u \sim v$ must be x or appear on the spine of the level-i cluster that contains x, and thus can be retrieved in $O(1)$ time. □

Now we turn to consider general $\mathcal{PM}_{i,j}$, for which we will develop a recursive strategy with multiple iterations. At each iteration, we pick a sequence $i = i_0 < i_1 < i_2 < \cdots < i_k = j$, for which $\mathcal{PM}_{i_0,i_1}, \mathcal{PM}_{i_1,i_2}, \ldots, \mathcal{PM}_{i_{k-1},i_k}$ are assumed to be solved

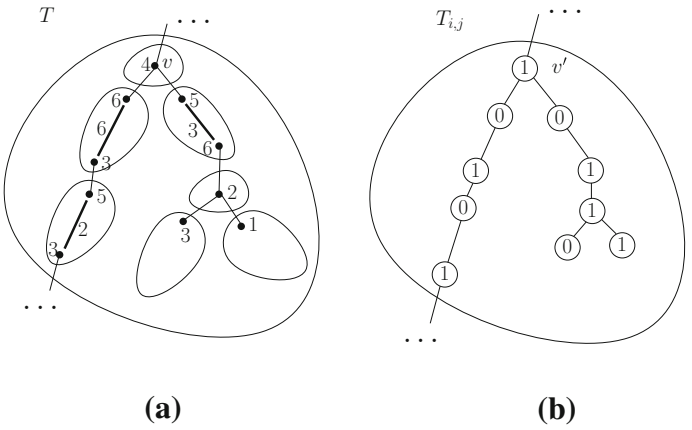

Fig. 5 An illustration for the proof of Lemma 3.6. **a** The *large splinegon* represents a level-j cluster and the *small* ones represent level-i clusters contained in the level-j cluster. *Bold lines* represent spines of level-i clusters. The number *alongside* a node is its weights, and the one *alongside* a spine is the minimum weight on the spine. **b** The 01-labeled tree $T_{i,j}$ that corresponds to the cluster head v

at the previous iteration. By Lemma 3.6, we solve $\mathcal{PM}'_{i,i_1}, \mathcal{PM}'_{i,i_2}, \ldots, \mathcal{PM}'_{i,i_k}$ using $O(k(5/6)^i n)$ bits of additional space and construction time.

Let u and v be the endpoints of a query path of $\mathcal{PM}_{i,j}$. That is, u and v are level-i boundary nodes that are contained in the same level-j cluster. As in the proof of Lemma 3.5, we still compute $t = \texttt{LCA}(\mathcal{B}, u, v)$ and partition the paths $u \sim t$ and $v \sim t$. For $u \sim t$, we compute $C_0 = \texttt{LLC}(\mathcal{B}, u, t)$, which is the lowest level cluster that contains both u and t. Let i' be the level of C_0. The case in which $i' = i$ can be simply handled by calling $\mathcal{PM}_{i_0,i_1}$. Otherwise, we determine s such that $i_s < i' \leq i_{s+1}$. Here s can be obtained in $O(1)$ time by precomputation for each possible value of i', which requires $O(\lg n)$ time and $O(\lg^2 n)$ bits of space. Let $C_1 = \texttt{level\_cluster}(\mathcal{D}, u, i_s)$, which is the level-$i_s$ cluster that contains u. Let $x = \texttt{cluster\_nn}(\mathcal{D}, C_1, t)$, which is a boundary node of C_1 that is between u and t. Similarly, let C_2 be the level-i_s cluster that contains t and let z be a boundary node of C_2 that is between u and t. By Lemma 3.2, x and z can be found in constant time. By Lemma 3.6, the paths $u \sim x$ and $z \sim t$ can be partitioned into $O(1)$ canonical paths by querying $\mathcal{PM}'_{i,i_s}$. Finally, the path $x \sim z$ can be partitioned recursively by querying $\mathcal{PM}_{i_s,i_{s+1}}$. See Fig. 6 for an illustration of partitioning $u \sim t$. On the other hand, the path $v \sim t$ can be partitioned in a similar fashion. Thus the partition of the path $u \sim v$ is obtained.

Summarizing the discussion above, we have the following recurrences. Here $S_\ell(i, j)$ is the space cost and the construction time, and $Q_\ell(i, j)$ is the query time spent at the first ℓ iterations for solving $\mathcal{PM}_{i,j}$. It should be drawn to the reader's attention that the coefficient of $Q_\ell(i_s, i_{s+1})$ is 1 in Equation 2, since a top-to-bottom query path requires at most one recursive call to subproblems of the form $\mathcal{PM}_{i_s,i_{s+1}}$.

$$S_{\ell+1}(i, j) = \sum_{s=0}^{k-1} S_\ell(i_s, i_{s+1}) + O(k(5/6)^i n) \tag{1}$$

$$Q_{\ell+1}(i, j) = \max_{s=0}^{k-1} Q_\ell(i_s, i_{s+1}) + O(1). \tag{2}$$

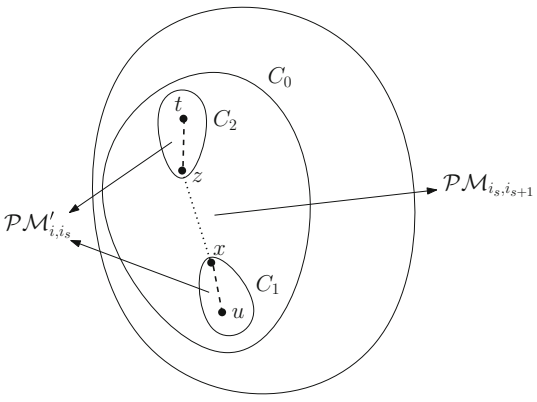

Fig. 6 An illustration of partitioning $u \sim t$. The *outermost splinegon* represents the level-i_{s+1} cluster that contains both u and t. The paths $u \sim x$ and $z \sim t$, which are represented by *dashed lines*, are partitioned by querying $\mathcal{PM}'_{i,i_s}$. The path $x \sim z$, which is represented by a *dotted line*, is partitioned by querying $\mathcal{PM}_{i_s,i_{s+1}}$

The desired recursive strategy follows from these recurrences.

Lemma 3.7 *Given a fixed value L, there exists a recursive strategy and some constant c such that, for $0 \le \ell \le L$, $S_\ell(i, A_\ell(i)) \le c(6/7)^i n$ and $Q_\ell(i, A_\ell(i)) \le c\ell$.*

Proof At the 0-th iteration, we set $A_0(i) = i + 1$. This can be used as the base case. By Lemma 3.5, $\mathcal{PM}_{i,i+1}$ can be supported using $O(1)$ query time at no extra space cost. Thus the statement holds for $\ell = 0$.

At the $(\ell + 1)$-st iteration, we choose the sequence $i, i + 13, A_\ell(i + 13), A_\ell^{(2)}(i + 13), \ldots, A_\ell^{(i)}(i + 13), A_\ell^{(i+1)}(i + 13)$. The last term is $A_{\ell+1}(i)$. By Equation 1, for some sufficiently large constant c_1:

$$\begin{aligned} S_{\ell+1}(i, A_{\ell+1}(i)) &\le \sum_{0\le j\le i} S_\ell(A_\ell^{(j)}(i + 13), A_\ell^{(j+1)}(i + 13)) + O(i(5/6)^i n) \\ &\le O(i(5/6)^i n) + \sum_{0\le j\le i} c_1(6/7)^{A_\ell^{(j)}(i+13)} \cdot n \\ &\le O(i(5/6)^i n) + \sum_{0\le j\le i} c_1(6/7)^{i+13+j} \cdot n \\ &\le O(i(5/6)^i n) + 7c_1(6/7)^{i+13} \cdot n \;\le\; c_1(6/7)^i \cdot n. \end{aligned}$$

This inequality follows because $7(6/7)^{13} \approx 0.9436 < 1$. Similarly, Equation 2 implies that, for some sufficiently large constant c_2,

$$\begin{aligned} Q_{\ell+1}(i, A_{\ell+1}(i)) &\le O(1) + \max_{0\le j\le i} Q_\ell(A_\ell^{(j)}(i + 13), A_\ell^{(j+1)}(i + 13)) \\ &\le O(1) + c_2\ell \le c_2(\ell + 1). \end{aligned}$$

The induction thus carries through, and the proof is completed by setting c to be the larger one of c_1 and c_2. □

We finally have Lemmas 3.8 and 3.9, which cover Theorem 1.1.

Lemma 3.8 *For $m \ge n$, $\mathcal{PM}_{0,h}$ can be solved using $O(\boldsymbol{\alpha}(m, n))$ query time in addition to $O(m)$ bits of extra space and construction time.*

Proof Given a parameter $m \ge n$, we set $L = \boldsymbol{\alpha}(m, n)$ and $h_\tau = 0$, and recurse one more iteration. At the final $(L + 1)$-st iteration, we pick the sequence $0, 1, 2, \ldots, \lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)$. The last term $A_L(\lfloor m/n \rfloor) > n \ge h$. This gives us

$$\begin{aligned} S_{L+1}(0, h) &\le S_{L+1}(0, A_L(\lfloor m/n \rfloor)) \\ &\le S_L(\lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)) + O(\lfloor m/n \rfloor n) \quad \text{(Equation 1 and Lemma 3.5)} \\ &\le O(m) \quad \text{(Lemma 3.7)} \end{aligned}$$

and

$$\begin{aligned} Q_{L+1}(0,h) &\le Q_{L+1}(0, A_L(\lfloor m/n \rfloor)) \\ &\le Q_L(\lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)) + O(1) \quad \text{(Equation 2 and Lemma 3.5)} \\ &\le O(L) \quad \text{(Lemma 3.7)} \\ &= O(\alpha(m,n)). \end{aligned}$$

Adding Lemmas 3.2 and 3.3, the overall space cost is $O(m)$ additional bits, the overall construction time is $O(m)$, and the query time is $Q_{L+1}(0,h) = O(\alpha(m,n))$. □

Lemma 3.9 *For $\mathrm{r}(n) = (6/7)^{\lg \alpha(n)} \cdot n = o(n)$, $\mathcal{PM}_{0,h}$ can be solved using $O(\alpha(n))$ query time, $2n + O(\mathrm{r}(n))$ bits of extra space, and $O(n)$ construction time.*

Proof We choose $L = \alpha(n)$ and $h_\tau = \lceil \lg L \rceil$. Note that $h_\tau = \omega(1)$ and $A_L(h_\tau) \ge h$. Therefore we have $S_L(h_\tau, A_L(h_\tau)) = O((6/7)^{h_\tau} n) = O(\mathrm{r}(n)) = o(n)$, and $Q_L(h_\tau, A_L(h_\tau)) = O(L) = O(\alpha(n))$. By Lemma 3.4, $\mathcal{PM}_{0,h_\tau}$ and $\mathcal{PM}'_{0,h_\tau}$ can be solved using $O(2^{h_\tau}) = O(\alpha(n))$ query time at no extra space cost. Adding Lemmas 3.2 and 3.3, the overall space cost is $2n + O(\mathrm{r}(n))$ additional bits, the overall construction time is $O(n)$, and the query time is $O(\alpha(n))$. □

By constructing the preorder label sequence [36,38] of T, we further have

Corollary 3.10 *Let T be an ordinal tree on n nodes, each having a weight drawn from $[1..\sigma]$. Then T can be represented (a) using $n \lg \sigma + O(m)$ bits of space to support path minimum queries in $O(\alpha(m,n))$ time, for any $m \ge n$; or (b) using $n(\lg \sigma + 2) + o(n)$ bits of space to support path minimum queries in $O(\alpha(n))$ time.*

By directly applying the result of Sadakane and Grossi [49], we can further achieve compression and replace the $n \lg \sigma$ additive term in the space cost of both the results of the above corollary by $nH_k + o(n) \cdot \lg \sigma$ while providing the same support for queries, where $k = o(\log_\sigma n)$ and H_k is the k-th order empirical entropy of the preorder label sequence.

4 Semigroup Path Sum Queries

In this section, we generalize the approach of supporting path minimum queries to semigroup path sum queries. As in Sect. 3, we transform the given tree into a binary tree, which is further decomposed using Lemma 2.1. We also define the notions of spines, heads and tails in the same manner. Again, our strategy is to make some paths canonical, for which the sum of weights along each canonical path will be precomputed and stored. Naturally, all singleton paths are still canonical. Each query path will be partitioned into disjoint canonical subpaths, and the sum of weights along the whole query path can be obtained by summing up the precomputed sums over these canonical subpaths.

Let $h_\tau > 0$ be a parameter whose value will be determined later. The spines of clusters whose levels are higher than or equal to h_τ are made canonical. As in Sect. 3, we define subproblems $\mathcal{PS}_{i,j}$ and $\mathcal{PS}'_{i,j}$ as follows:

- $\mathcal{PS}_{i,j}$: sum up weights of nodes along query paths between two level-i boundary nodes that are contained in the same level-j cluster;
- $\mathcal{PS}'_{i,j}$: sum up weights of nodes along query paths from a level-i boundary node to a level-j one, where both boundary nodes are contained in the same level-j cluster.

$\mathcal{PS}_{0,h_\tau}$ will be solved using brute-force search, while $\mathcal{PS}_{h_\tau,h}$ will be solved using the recursive approach as described in Sect. 3. In the following, Lemmas 4.1 to 4.4 are modified from Lemmas 3.3 to 3.6. For each lemma, we give its proof only if the proof for the corresponding lemma in Sect. 3 cannot be applied directly.

Lemma 4.1 *It requires $O((5/6)^{h_\tau} n \lg \sigma)$ bits of additional space and $O(n)$ construction time to make these spines canonical.*

Proof As the semigroup contains σ elements, it requires $\lg \sigma$ bits to store the sum of weights on the spine of a cluster. Thus the overall space cost is $\sum_{i=h_\tau}^{h}((5/6)^i \cdot 2n \cdot \lg \sigma) = O((5/6)^{h_\tau} n \lg \sigma)$ bits. In particular, when $h_\tau = \omega(1)$, the space cost is $o(n \lg \sigma)$ bits. The construction is the same in Lemma 3.3. □

Lemma 4.2 *$\mathcal{PS}_{0,h_\tau}$ can be solved using $O(2^{h_\tau})$ query time and no extra space.*

Lemma 4.3 *For every pair of i and j satisfying that $h_\tau \le i < j \le h$ and $j - i = O(1)$, $\mathcal{PS}_{i,j}$ can be solved using $O(1)$ query time and no extra space.*

Lemma 4.4 *For a fixed pair of i and j satisfying that $h_\tau \le i < j \le h$, $\mathcal{PS}'_{i,j}$ can be solved using $O(1)$ query time, $O((5/6)^i n \lg \sigma)$ bits of extra space, and $O((5/6)^i n)$ construction time.*

Proof We make each query path of $\mathcal{PS}'_{i,j}$ canonical and store the sum of weights along each of these paths explicitly. It is easy to see that the space cost is $O((5/6)^i n \lg \sigma)$ extra bits and the construction time is $O((5/6)^i n \lg \sigma)$. □

Following the same recursive strategy in Sect. 3, we have the following recurrences. Here $S_\ell(i, j)$ is the space cost, $P_\ell(i, j)$ the construction time, and $Q_\ell(i, j)$ is the query time spent at the first ℓ iterations for solving $\mathcal{PS}_{i,j}$.

$$S_{\ell+1}(i, j) = \sum_{s=0}^{k-1} S_\ell(i_s, i_{s+1}) + O(k(5/6)^i n \lg \sigma) \tag{3}$$

$$P_{\ell+1}(i, j) = \sum_{s=0}^{k-1} P_\ell(i_s, i_{s+1}) + O(k(5/6)^i n) \tag{4}$$

$$Q_{\ell+1}(i, j) = \max_{s=0}^{k-1} Q_\ell(i_s, i_{s+1}) + O(1). \tag{5}$$

We then have the following key lemma, which is similar to Lemma 3.7.

Lemma 4.5 *Given a fixed value L, there exists a recursive strategy and some constant c such that, for $0 \le \ell \le L$, $S_\ell(i, A_\ell(i)) \le c(6/7)^i n \lg \sigma$, $P_\ell(i, A_\ell(i)) \le c(6/7)^i n$, and $Q_\ell(i, A_\ell(i)) \le c\ell$.*

Finally, we store weights of nodes in the preorder label sequence [36,38] of T. This requires $n \lg \sigma + o(n)$ bits of space, and the weight of each node can be accessed in $O(1)$ time. The rest of the proof for Theorem 1.2 follows from the same strategies of Lemmas 3.8 and 3.9.

5 Encoding Topology Trees: Proof of Lemma 3.2

Let T be an ordinal tree on n nodes. As described in Sect. 3, we transform T into a binary tree $\mathcal{B}$, and compute the directed topology tree of $\mathcal{B}$ as $\mathcal{D}$. Let $n_{\mathcal{D}}$ denote the number of nodes in $\mathcal{D}$. By Lemma 2.1, we have that $n_{\mathcal{D}} = O(n)$, as there are at most $(5/6)^i \cdot 2n$ level-i clusters. Let $i_1 = \lceil 12 \lg \lg n \rceil$ and $i_2 = \lfloor \lg \lg n \rfloor - 1$. Again by Lemma 2.1, there are at most $n_1 = (5/6)^{i_1} \cdot n = O(n/(6/5)^{12 \lg \lg n}) = O(n/\lg^{12 \lg(6/5)} n) < O(n/\lg^3 n)$ level-i_1 clusters, each being of size at most $m_1 = 2^{i_1} \leq 2^{12 \lg \lg n + 1} = 2 \lg^{12} n$. Similarly, there are at most $n_2 = (5/6)^{i_2} \cdot n = O(n/\lg^{\lg(6/5)} n) < O(n/(\lg^{1/5} n))$ level-i_2 clusters, each being size of at most $m_2 = 2^{i_2} \leq 2^{\lg \lg n - 1} = (\lg n)/2$. Clusters at levels i_1 and i_2 are referred to as *mini-clusters* and *micro-clusters*, respectively.

We will precompute several lookup tables that support certain queries for each possible micro-cluster. Note that two clusters are different if the sets of non-dummy nodes are different. There are $O(n^{1-\delta})$ distinct micro-clusters for some $\delta > 0$, since $m_2 \leq (1/2) \lg n$. If each micro-cluster costs $o(n^{\delta})$ bits, then the space cost of the lookup table is only $o(n)$ additional bits. We first make use of a lookup table to store the encodings of micro-clusters.

Lemma 5.1 *All micro-clusters can be encoded in $2n + o(n)$ bits of space such that given the topological rank of a micro-cluster, its encoding can be retrieved in $O(1)$ time.*

Proof Note that $\mathcal{B}$ has at most $2n$ nodes. Given a micro-cluster C, we do not store its encoding directly because it could require about $4n$ bits of space for all micro-clusters. Instead, we define X to be the union of non-dummy nodes and dummy boundary nodes of C and store only C_X, where C_X is the X-extraction of C as defined in Sect. 2.2. We also mark the (at most 2) dummy nodes in C_X, which requires $O(\lg m_2) = O(\lg \lg n)$ bits per node. As illustrated in Fig. 7, we encode C_X as *balanced parentheses* [44]. The overall space cost of encoding C is $2n_C + O(\lg \lg n)$ bits, where n_C is the number of non-dummy nodes in C. We concatenate the above encodings of all micro-clusters ordered by topological rank and store them in a sequence, P, of $n' = 2n + O(n \lg \lg n/(\lg^{1/5} n))$ bits.

We construct a sparse bit vector, P', of the same length, and set $P'[i]$ to 1 iff $P[i]$ is the first bit of the encoding of a micro-cluster. P' can be represented using Lemma 2.2 in $\lg \binom{n'}{n_2} + O(n \lg \lg n/\lg n) = O(n \lg \lg n/(\lg^{1/5} n))$ bits to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in constant time. We construct another bit vector $B_0[1..n_{\mathcal{D}}]$, in which $B_0[j] = 1$ iff the cluster with topological rank j is a micro-cluster, which is also encoded using Lemma 2.2 in $O(n \lg \lg n/(\lg^{1/5} n))$ bits.

To retrieve the encoding of a cluster, C, whose topological rank is j, we first use B_0 to check if C is a micro-cluster. If this is true, let $r = \texttt{rank}_1(B_0, j)$. Then the

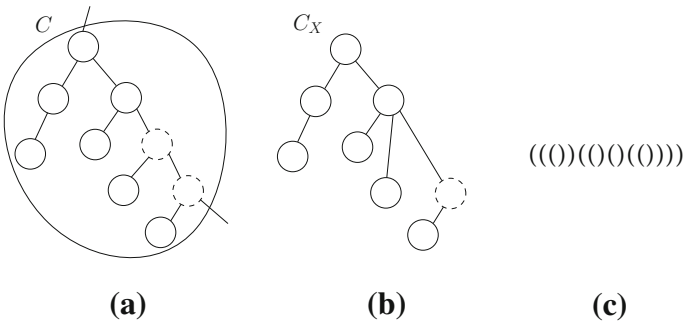

Fig. 7 An example of encoding micro-clusters. **a** A micro-cluster C in which dummy nodes are represented by *dashed circles*. **b** The corresponding C_X obtained by preserving non-dummy nodes and dummy boundary nodes. **c** The balanced parentheses for C_X

encoding of C_X is $P[\texttt{select}_1(P', r)..\texttt{select}_1(P', r+1)-1]$. To recover C from C_X, we need only to reverse the binary tree transformation described at the beginning of Sect. 3.2. This can be done in $O(1)$ time using a lookup table F_0 of $o(n)$ bits. □

Now we start to consider the support for operations. By Lemma 2.1, each node is contained in exactly one cluster at each level. We borrow the terminology from Geary et al.'s work [30] and define the τ-*name* of a node x to be $(\tau_1(x), \tau_2(x), \tau_3(x))$, where $\tau_1(x)$, $\tau_2(x)$ and $\tau_3(x)$ are the topological ranks of the level-i_1, level-i_2, and level-0 clusters that contain x, respectively. Let $i_3 = 0$. Note that, for $k = 1, 2$, $\tau_{k+1}(x)$ is represented as the relative value with respect to the level-i_k cluster that contains x, i.e., the difference between the topological ranks of the level-i_{k+1} and level-i_k clusters that contain x. Thus $\tau_2(x)$ and $\tau_3(x)$ can be encoded in $O(\lg \lg n)$ bits.

As in Lemma 2.1, preorder segments are defined to be maximal substrings of nodes in the preorder sequence that are in the same cluster [34, Definition 4.22]. By the same lemma, each cluster contains only one node or has only one child cluster, and thus the nodes of each cluster belong to at most 2 *preorder segments*. These preorder segments and the cluster are said to be *associated with* each other, and the preorder segments of a level-i cluster are called *level-i preorder segments*. For simplicity, level-i_1 and level-i_2 preorder segments are also called *mini-segments* and *micro-segments*, respectively.

In the following proofs, we will precompute several lookup tables that store certain information for each possible micro-cluster C. When we say *the hierarchy of C*, we mean the hierarchy obtained by partitioning C as described in Lemma 2.1.

Lemma 5.2 *It requires $O(1)$ time and $o(n)$ bits of additional space for the conversion between the preorder rank of a node in T or $\mathcal{B}$ and its τ-name.*

Proof We only consider the conversion for nodes in $\mathcal{B}$; the other case can be handled similarly. For each level $i \in [i_2..h]$ and each level-i cluster C, we store the following information in $D(C)$: its topological rank, its root node, its boundary nodes, and the starting and ending positions of its associated preorder segments in $\mathcal{B}$. For levels i_1 to h, it requires $O(\lg n)$ bits of space per cluster to store the information directly (the nodes stored in $D(C)$ are encoded as their preorder ranks in $\mathcal{B}$). For each cluster C at

levels i_2 to $i_1 - 1$, we store the relative ranks with respect to the mini-cluster C' that contains C. More precisely, we encode the difference between the topological ranks of C and C', and each node stored in $D(C)$ is encoded as i if it is the i-th node in C' in preorder. This requires only $O(\lg \lg n)$ bits per cluster, as each mini-cluster is of size at most $m_1 = O(\lg^{12} n)$. Physically, for all clusters above level i_2, we use two arrays to store the above information and use two bit vectors to locate the corresponding entry for any given cluster. For levels i_1 to h, we construct a bit vector $B_1[1..n_{\mathcal{D}}]$ in which $B_1[j] = 1$ iff the cluster with topological rank j is at or above level i_1. We construct an array D_1 whose length is equal to the number of clusters at levels i_1 to h; for each cluster C at levels i_1 to $h - 1$, $D(C)$ is stored in the $\texttt{rank}_1(B_1, j)$-th entry of D_1 if the topological rank of C is j. Similar auxiliary data structures D_2 and B_2 are also constructed for clusters at levels i_2 to $i_1 - 1$. In the hierarchy $\mathcal{D}$, there are $O(n_1)$ clusters at levels i_1 to h, and $O(n_2)$ clusters at levels i_2 to $i_1 - 1$. Thus the overall space cost, including the cost of encoding B_1 and B_2 using Lemma 2.2, is $O(n_1) \times O(\lg n) + O(n_2) \times O(\lg \lg n) = o(n)$ bits.

For $k \in \{1, 2\}$, all level-i_k preorder segments form a partition of the preorder traversal sequence of $\mathcal{B}$, and we mark their starting positions in a bit vector V_k. More precisely, we set $V_k[j] = 1$ if and only if the j-th node in preorder of $\mathcal{B}$ is the first node in some level-i_k preorder segment. By Lemma 2.2, these two bit vectors can be encoded in $o(n)$ bits of space to support `rank` and `select` in $O(1)$ time. For mini-segment (or micro-segment) s, we store in $E(s)$ the topological rank of its associated mini-cluster (or micro-cluster). It requires $O(n_1) \times O(\lg n) = O(n/\lg^2 n)$ bits to store $E(s)$ directly for all mini-segments. For each micro-segment s, we store the relative topological rank of its associated micro-cluster with respect to the mini-cluster that contains s. This requires only $O(\lg \lg n)$ bits per micro-segment, and $O(n_2) \times O(\lg \lg n) = O(n \lg \lg n/(\lg^{1/5} n))$ bits in total. See Fig. 8 for an example.

Finally, we precompute a lookup table F_1 that stores, for each possible micro-cluster C, the mapping between nodes in C and level-0 clusters in the i_2-level hierarchy of C. These nodes and level-0 clusters are encoded as the relative preorder ranks and topological ranks with respect to C. It is easy to see that F_1 occupies $o(n)$ bits of space, since $m_2 \leq (1/2) \lg n$.

Given a node x in $\mathcal{B}$, we first locate s_1, the mini-segment that contains x, using `rank` operations on V_1. By accessing $E(s_1)$, we can determine C_1, which is the mini-cluster that contains x and the associated mini-segment s_1. Then we locate s_2, the micro-segment that contains x, using V_2. By accessing $E(s_2)$ and $D(C_1)$, we can determine C_2, the micro-cluster that contains x. Finally, by accessing F_1, we can find the level-0 cluster that represents x. That is, we obtain the τ-name of x.

In the other direction, given the τ-name of some node x, we can immediately determine C_1, C_2 and C_3, which are the level-i_1, level-i_2 and level-0 clusters that contain x, respectively. By accessing $D(C_1)$ and $D(C_2)$, we can find the associated preorder segments of C_2. By accessing F_1, we can compute the relative preorder rank of x with respect to C_2. Then we can determine the preorder rank of x in $\mathcal{B}$. □

The conversion between nodes in T and $\mathcal{B}$ directly follows from Lemma 5.2. Thus, to answer queries that ask for a node, it suffices to return either its τ-name, or its preorder rank in $\mathcal{B}$ or T. In the remaining part of this section, when we talk about the

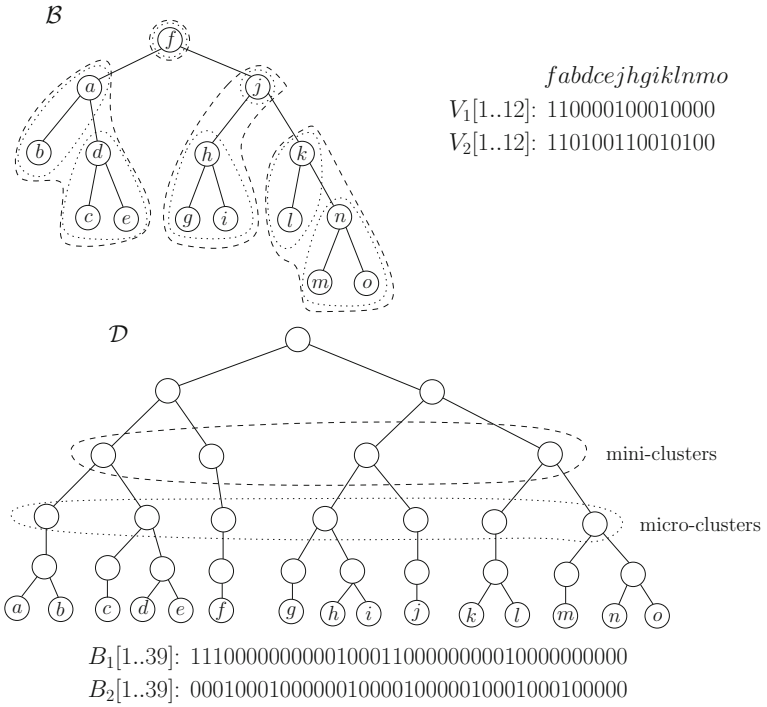

Fig. 8 An illustration for the proofs of Lemmas 5.2 and 5.3. Here mini-clusters and micro-clusters are enclosed by *dashed* and *dotted splinegons*, respectively. The bit vectors B_1, B_2, V_1, and V_2 are constructed for the directed topology tree shown in Fig. 3

preorder rank of a node x, we refer to the preorder rank of x in $\mathcal{B}$, unless otherwise specified.

Lemma 5.3 *Operations* `cluster_head` *and* `cluster_tail` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof We precompute a lookup table F_2 that stores, for each possible micro-cluster C and each cluster C' in the i_2-level hierarchy of C, the root and the boundary nodes of C'. These nodes are encoded as the relative preorder ranks with respect to C. It is clear that F_2 occupies $o(n)$ bits of space.

Recall the bit vectors B_1 and B_2 constructed in the proof of Lemma 5.2, as well as the information stored in $D(C)$ for each cluster C whose level is higher than or equal to i_2 in the same proof. Given a cluster C with topological rank j, we first determine if the level of C is at or above i_1 by checking if $B_1[j] = 1$. If the level of C is at or above i_1, then we can retrieve the answers to `cluster_head` and `cluster_tail` directly from $D(C)$. Otherwise, we find the mini-cluster C_1 that contains C by $\texttt{select}_1(B_1, \texttt{rank}_1(B_1, j))$. Then we determine if the level of C is in $[i_2..i_1 - 1]$ by checking if $B_2[j] = 1$. If this is true, then we can obtain the answers using $D(C_1)$ and $D(C)$: To locate the root, c_r, of C (the boundary nodes can be located using the same approach), the relative preorder of c_r in C_1 can be

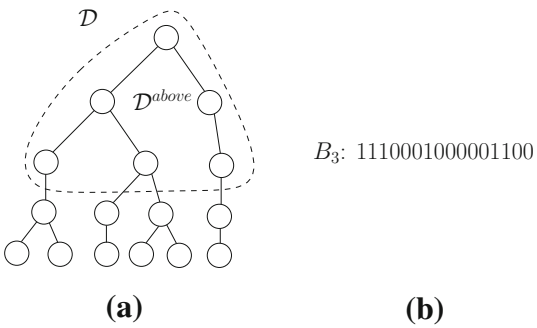

Fig. 9 An illustration of the support for `level_cluster`. **a** A directed topology tree $\mathcal{D}$ in which the topmost three levels belong to $\mathcal{D}^{above}$. **b** The corresponding bit vector B_3 for $\mathcal{D}$ and $\mathcal{D}^{above}$

used to locate the preorder segment in C_1 containing c_r. The preorder rank of the first node in this segment, which is stored in $D(C_1)$, can be used to further compute the preorder rank of c_r in constant time. If the level of C is not within the above range, we further find the micro-cluster C_2 that contains C by $\texttt{select}_1(B_2, \texttt{rank}_1(B_2, j))$. By accessing $D(C_1)$, $D(C)$ and the entry in F_2 that corresponds to the encoding of C_2, we can obtain the answers in $O(1)$ time. □

Lemma 5.4 *Operations* `level_cluster` *and* `LLC` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof Let $\mathcal{D}^{above}$ be the topmost $h - i_2 + 1$ levels of $\mathcal{D}$. It is clear that $\mathcal{D}^{above}$ represents the hierarchy for the clusters at levels i_2 to h. We store $\mathcal{D}^{above}$ using Lemma 2.5. This requires $O(n_2) = O(n/(\lg^{1/5} n))$ bits of space, since there are at most $O(n_2)$ clusters at these levels. As illustrated in Fig. 9, we construct a bit vector B_3 in which $B_3[j]$ is 1 iff the cluster with topological rank j is present in $\mathcal{D}^{above}$. By Lemma 2.2, B_3 occupies $o(n)$ bits of space and can be used to perform conversions between the topological rank of any given cluster present in $\mathcal{D}^{above}$ and its preorder rank in $\mathcal{D}^{above}$ in constant time.

Let n'_C be the number of nodes in a micro-cluster C, including both dummy and non-dummy ones. We precompute a lookup table F_3 that stores, for each possible micro-cluster C, each level $i \in [0..i_2]$, and each $j \in [1..n'_C]$, the relative topological rank of the level-i cluster in the hierarchy of C that contains the j-th node of C. We also precompute another lookup table F_4 that stores, for each possible micro-cluster C, and each $j_1, j_2 \in [1..n'_C]$, the lowest level cluster in the hierarchy of C that contains the j_1-st and the j_2-nd nodes of C.

We have two cases in computing $\texttt{level\_cluster}(\mathcal{D}, i, x)$. If $i \leq i_2$, then the answer can be retrieved from F_3 directly. Otherwise, we can determine the micro-cluster that contains x using the τ-name of x, and find the level-i cluster that contains x using `level_anc` operations on $\mathcal{D}^{above}$.

The support for $\texttt{LLC}(\mathcal{D}, x, y)$ is similar. We first determine if x and y are in the same micro-cluster using their τ-names. If they are in the same micro-cluster, then the answer can be retrieved from F_4 in $O(1)$ time. Otherwise, we can find the micro-clusters that contain x and y, respectively. Let these micro-clusters be C_1 and C_2. We can compute `LLC` by finding the lowest common ancestor of C_1 and C_2 in $\mathcal{D}^{above}$. □

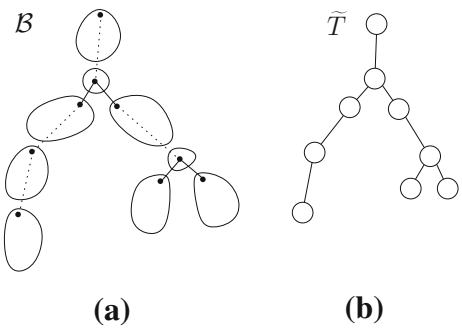

Fig. 10 An illustration for the proof of Lemma 5.5. **a** A binary tree $\mathcal{B}$ in which micro-clusters are represented by *splinegons*, and their roots are represented by *solid circles*. **b** The tree $\widetilde{T}$ obtained by extracting roots of micro-clusters in $\mathcal{B}$

Lemma 5.5 *Operation* `LCA` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof As described in Sect. 2.2, we extract the roots of all micro-clusters from $\mathcal{B}$. As illustrated in Fig. 10, the extracted tree, $\widetilde{T}$, is maintained using Lemma 2.5. For the conversion between nodes in $\widetilde{T}$ and roots of micro-clusters in $\mathcal{B}$, we maintain a bit vector V_3 using Lemma 2.2, for which the j-th bit is 1 iff the j-th node of $\mathcal{B}$ is the root of some micro-cluster. In addition, we precompute a lookup table F_5 that stores, for each possible micro-cluster C, and each $j_1, j_2 \in [1..n'_C]$, the lowest common ancestor of the j_1-st and the j_2-nd nodes of C.

To compute $\texttt{LCA}(\mathcal{B}, x, y)$, we first verify if nodes x and y are in the same micro-cluster. If they are both in the same cluster, then their lowest common ancestor can be found by accessing F_5. Otherwise, we determine the micro-clusters that contain x and y. Let u and v be the roots of these micro-clusters, respectively. We perform an `LCA` operation on $\widetilde{T}$ and find the lowest micro-cluster root that is a common ancestor of u and v. This root must be the lowest common ancestor of x and y. □

Lemma 5.6 *Operation* `cluster_nn` *can be supported in* $O(1)$ *query time and no extra space.*

Proof Let C be a cluster and let x be some node that is outside of C. To compute $\texttt{cluster\_nn}(\mathcal{D}, C, x)$, we first determine if x is in the subtree rooted at the root of C. Let $y = \texttt{cluster\_head}(\mathcal{D}, C)$. If $\texttt{LCA}(\mathcal{B}, x, y) \neq y$, then x is also outside of the subtree rooted at y, and the closest boundary node to x must be y. Otherwise, the closest boundary node is $z = \texttt{cluster\_tail}(\mathcal{D}, C)$, the tail of the cluster C. □

Lemma 5.7 *Operation* `parent` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof We precompute a lookup table F_6 that stores, for each possible micro-cluster C and each $j \in [1..n'_C]$, the parent of the j-th node of C. Given a node x, we first find the micro-cluster C that contains x. If x is not the root of C, then we can retrieve the parent node of x from F_6 directly. Otherwise, we find the lowest micro-cluster C' above C by finding the parent of the node in $\widetilde{T}$ that represents C. Then the parent of x must be the closest boundary node of x in C', which can be found by `cluster_nn`. □

Lemma 5.8 *Operations* `BN_rank` *and* `BN_select` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof We first construct data structures to support `BN_rank`$(\mathcal{B}, i, x)$ and `BN_select` $(\mathcal{B}, i, j)$ when $i \in [i_2..h]$. For each $i \in [i_2..h]$, we construct a bit vector I_i, in which $I_i[j] = 1$ iff the j-th node in preorder is a level-i boundary node. Each of these $O(h) = O(\lg n)$ bit vectors is encoded using Lemma 2.2. It is clear that to support these two operations in this special case, it is sufficient to perform a single `rank` or `select` operation on I_i, and thus can be performed in constant time. We calculate the space cost in two steps. We first sum up the space cost of all I_i's for $i \in [i_1, h]$. The number of 1-bits in each of these bit vectors is $O(n/\lg^3 n)$, and thus the total cost of these $O(\lg n)$ bit vectors is $O(\lg n) \times O(n \lg\lg n/\lg^3 n) = O(n \lg\lg n/\lg^2 n)$. We then sum up the space occupancy of all I_i's for $i \in [i_2..i_1 - 1]$. As the number of 1 bits in each of these bit vectors is $O(n/\lg^{1/5} n)$, the total cost of these $O(\lg\lg n)$ bit vectors is $O(n(\lg\lg n)^2/\lg^{1/5} n)$ bits, which subsumes the space cost computed in the previous step.

Next we construct data structures to support `BN_rank`$(\mathcal{B}, i, x)$ for $i \in [0..i_2 - 1]$. Let n_s and n'_s denote the number of mini-segments and micro-segments, respectively. We construct the following two sets of arrays and one lookup table:

- An array $G_i[1..n_s]$ for each $i \in [0..i_2 - 1]$, in which $G_i[j]$ stores the number of level-i boundary nodes that precede the j-th mini-segment in preorder;
- An array $G'_i[1..n'_s]$ for each $i \in [0..i_2 - 1]$, in which $G'_i[j]$ stores the number of level-i boundary nodes that precede the j-th micro-segment in preorder and also reside in the same mini-segment containing this micro-segment;
- A universal lookup table F_7 that stores, for any possible micro-cluster C (here micro-clusters with the same tree structure but different micro-segments are considered different; recall that there are at most 2 micro-segments in each micro-cluster), any node, x, in C identified by its τ_3-name, and any number $i \in [0..i_2 - 1]$, the number of level-i boundary nodes preceding x in preorder.

We illustrate the definition of G_i's in Fig. 11. To analyze storage cost, observe that the space costs of all the G'_i's dominate the overall cost, which is $O(\lg\lg n) \times O(n \lg\lg n/\lg^{1/5} n) = O(n(\lg\lg n)^2/\lg^{1/5} n)$ bits.

With these auxiliary structures and the bit vectors V_1 and V_2 constructed in the proof of Lemma 5.2, we can support `BN_rank`$(\mathcal{B}, i, x)$ for $i \in [0..i_2 - 1]$ as follows. We first retrieve $G_i[\texttt{rank}(V_1, i)]$ which is the number of level-i boundary nodes preceding the mini-segment, s, containing x, and $G'_i[\texttt{rank}(V_2, i)]$ which is the number of level-i boundary nodes inside s that precede the micro-segment, s', containing x. It now suffices to compute the number of level-i boundary nodes preceding x inside s'. This can be computed by first retrieving the encoding of the micro-cluster containing x (its topological rank is $\tau_1(x) + \tau_2(x)$) and then perform a table lookup in F_7.

Finally, to support `BN_select`$(\mathcal{B}, i, j)$ for $i \in [0..i_2 - 1]$, we construct the following data structures (e_i denotes the number of level-i boundary nodes):

- A bit vector $R_i[1..e_i]$ for each $i \in [0..i_2 - 1]$, in which $R_i[j] = 1$ iff the j-th level-i boundary node in preorder has the smallest preorder rank among all the level-i

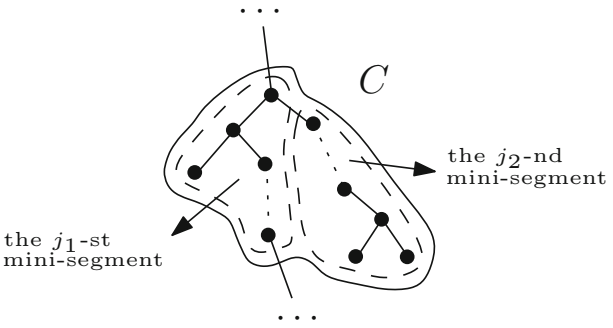

Fig. 11 An illustration of the support for `BN_rank`. Here a mini-cluster C with two mini-segments is enclosed by a *solid splinegon*, and the nodes of these two mini-segments are enclosed by *dashed splinegons*. We draw only level-i boundary nodes inside C and the edges and paths that connect them, which are represented by *solid* and *dotted lines*, respectively. Then we have $G_i[j_1] = k_1$, $G_i[j_1 + 1] = k_1 + 5$, $G_i[j_2] = k_1 + k_2 + 5$, and $G_i[j_2 + 1] = k_1 + k_2 + 10$, where k_1 is the number of level-i boundary nodes that precede the head of C in preorder, and k_2 is the number of level-i boundary nodes that are descendants of the tail of C

boundary nodes contained in the same mini-segment (let t_i denote the number of 1-bits in R_i);

- An array $S_i[1..t_i]$, in which $S_i[j]$ stores the topological rank of the mini-cluster containing the level-i boundary node corresponding to the j-th 1-bit in R_i;
- A bit vector $R'_i[1..e_i]$ for each $i \in [0..i_2 - 1]$, in which $R'_i[j] = 1$ iff the j-th level-i boundary node in preorder has the smallest preorder rank among all the level-i boundary nodes contained in the same micro-segment (let t'_i denote the number of 1-bits in R'_i);
- An array $S'_i[1..t'_i]$, in which $S'_i[j]$ stores a pair: the first item is the relative topological rank of the micro-cluster (relative to the mini-cluster that it resides in) containing the level-i boundary node y corresponding to the j-th 1-bit in R'_i, and an integer in $\{1, 2\}$ indicating which of the up to 2 micro-segments inside this micro-cluster contains y;
- A universal lookup table F_8 that stores, for any possible micro-cluster C, any number $i \in [0..i_2 - 1]$, any number $j \in [1..n_C]$ and any number $k \in \{1, 2\}$, the τ_3-name of the j-th level-i boundary node in the k-th micro-segment of C, or -1 if such a node does not exist.

As $t_i = O(n/\lg^3 n)$ and $t'_i = O(n/\lg^{1/5} n)$ for $i \in [0..i_2 - 1]$, it is easy to show that these structures occupy $O(n(\lg\lg n)^2/\lg^{1/5} n)$ bits. To support `BN_select`$(\mathcal{B}, i, j)$ for $i \in [0..i_2 - 1]$, let z denote the answer to be computed. We first use R_i and S_i to locate the mini-cluster containing z, and then use R'_i and S'_i to locate z's micro-cluster C. In this process, we also find out which micro-segment of C contains z, and how many level-i boundary nodes precede z in preorder are in the same micro-segment. A table lookup using F_8 will complete this process. □

6 Path Reporting Queries

In this section, we consider the problem of supporting path reporting queries. Let T denote the input tree. We represent T as an ordinal one, and assume that the weights of nodes are drawn from $[1..\sigma]$. We follow the general strategy of He et al. [36] that makes use of range trees and tree extraction. To achieve new results, we also make novel use of several other data structural techniques, including tree extraction described in Sect. 2.2, the ball-inheritance problem described in Sect. 2.3, and finally the succinct indices developed in Sect. 3.

For completeness, we review the data structures of He et al. [36]. We build a conceptual range tree on $[1..\sigma]$ with the branching factor $f = \lceil \lg^{\epsilon} n \rceil$. Starting from the top level, which contains $[1..\sigma]$ initially, we keep splitting each range at the current lowest level into f child ranges of almost equal sizes, until we obtain σ leaf ranges that contains a single weight each. This conceptual range tree has $h = \lceil \log_f \sigma \rceil + 1$ levels, which are numbered from top to bottom. The top level is the first level, and the bottom level is the h-th level.

For $\ell = 1, 2, \ldots, h-1$, we create an auxiliary tree T_ℓ for the ℓ-th level, which initially contains a dummy root r_ℓ only. We list the ranges at the ℓ-th level in increasing order of left endpoints. Let $[a_1..b_1], [a_2..b_2], \ldots, [a_m..b_m]$ be these ranges. For $i = 1, 2, \ldots, m$, we construct F_{a_i,b_i} as described in Sect. 2.2, and add the roots of ordinal trees in F_{a_i,b_i} as children of r_ℓ, preserving the original left-to-right order. Remember that the ranges at each level form a disjoint union of $[1..\sigma]$. Thus there is a one-to-one correspondence between the non-dummy nodes in T_ℓ and the nodes in T.

For each T_ℓ, we assign labels to its nodes. The dummy root is always assigned 1. For each node x in T, we use x_ℓ to denote the node in T_ℓ that corresponds to x. We say that a range $[a..b]$ at the ℓ-th level contains x_ℓ if the weight of x is between a and b. We assign a label α to x_ℓ if the range at the $(\ell+1)$-st level that contains $x_{\ell+1}$ is the α-th child of the range at the ℓ-th level that contains x_ℓ. See Fig. 12 for an example. T_ℓ is maintained using a succinct representation for labeled ordinal trees over a sublogarithmic alphabet, which is summarized in Lemma 6.1.

Lemma 6.1 ([36, Lemma 6]). *An ordinal tree T on n nodes whose labels are drawn from $[1..f]$, where $f = O(\lg^{\epsilon} n)$, can be represented using $n(\lg f + 2) + o(n)$ bits of space to support the operations listed in Lemma 2.5 and the following operations in constant time. Here x and z are nodes in T, α and β are in $[1..f]$, and a node whose label is α is said to be an α-node.*

- `pre_rank`$_\alpha(T, x)$ *Return the number of α-nodes that precede x in preorder;*
- `pre_select`$_\alpha(T, i)$ *Return the i-th α-nodes in preorder of T;*
- `pre_count`$_\beta(T, i)$ *Return the number of nodes whose preorder ranks are at most i and labels are at most β;*
- `lowest_anc`$_\alpha(T, x)$ *Return the lowest α-ancestor of x if such an α-ancestor exists, otherwise return* `NULL`*;*
- `node_summarize`(T, x, z) *Given that node z is an ancestor of x, this operation returns f bits, where the α-th bit is 1 if and only if there exists an α-node on the path from x to z (excluding z), for $1 \le \alpha \le f$.*

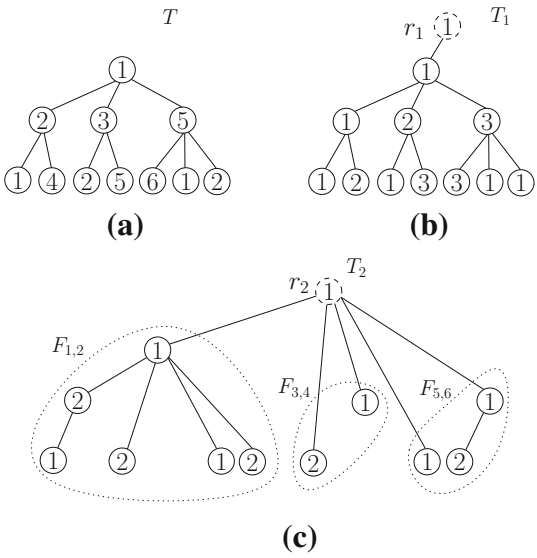

Fig. 12 **a** An input tree T with $n = 11$ and $\sigma = 6$, for which the conceptual range tree has branching factor $f = 3$. **b** The corresponding tree T_1, where the dummy root r_1 is represented by a *dashed circle*. **c** The corresponding tree T_2, where the dummy root r_2 is represented by a *dashed circle*, and $F_{1,2}$, $F_{3,4}$, and $F_{5,6}$ are marked by *dotted splinegons*

When representing each T_ℓ using Lemma 6.1, the preorder label sequence of T_ℓ is stored explicitly. Thus the preorder label sequences of all T_ℓ's essentially form the generalized wavelet tree of the preorder label sequence of T. The $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$, $\texttt{pre\_count}_\beta$ and $\texttt{lowest\_anc}_\alpha$ operations allow us to traverse up and down this generalized wavelet tree using standard wavelet tree algorithms. More details are given by He et al. [36], and their result is summarized in the following lemma:

Lemma 6.2 ([36]). *Given a node x in T_ℓ, where $1 < \ell \leq h - 1$, its corresponding node in $T_{\ell-1}$ can be found in $O(1)$ time. Similarly, given a node x in T_ℓ, where $1 \leq \ell < h - 1$, its corresponding node in $T_{\ell+1}$ can be found in $O(1)$ time.*

We also store one variant of the auxiliary data structures described in Lemma 2.3, which support the ball-inheritance problem using $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of space and $O(\mathtt{t}(\sigma))$ query time. This implies the following lemma:

Lemma 6.3 *Given a node x in T_ℓ, where $1 \leq \ell \leq h - 1$, its corresponding node in T can be found using $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of additional space and $O(\mathtt{t}(\sigma))$ time.*

Now we describe the details of achieving improved query time. Let u and v denote the endpoints of the query path, and let $[p..q]$ denote the query range. We compute $t = \texttt{LCA}(u, v)$. Let $A_{u,t}$ denote the set of nodes on the path from u to and excluding t. Thus the query path can be decomposed into $A_{u,t}$, $A_{v,t}$, and $\{t\}$. We only consider how to report nodes in $A_{u,t} \cap R_{p,q}$, where $R_{p,q}$, as defined in Sect. 2.2, is the set of nodes in T whose weights are in $[p..q]$.

We find the lowest range in the conceptual range tree that covers the query range $[p..q]$. This range, which is denoted by $[a..b]$, can be computed from the lowest common ancestor of the leaf ranges containing p and q. Let k denote the level that

contains $[a..b]$. We then locate the nodes x and z in T_k that correspond to $anc_{a,b}(T, u)$ and $anc_{a,b}(T, t)$, respectively.

Lemma 6.4 *The nodes x and z can be found using $O(n \lg \sigma)$ bits of additional space and $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time.*

Proof These two nodes can be found using either of the following two approaches. The first approach applies Lemma 6.2 repeatedly, which requires $O(k) = O(h) = O(\lg\sigma/\lg\lg n + 1)$ time.

The second approach is described as follows. For the non-dummy nodes in each $F_{a,b}$, we list the preorder ranks of their corresponding nodes in T as a conceptual array $S_{a,b}$. We maintain $S_{a,b}$ using succinct indices for predecessor search [32], which require $O(\lg\lg n)$ bits per entry and support predecessor and successor queries in $O(\lg\lg n)$ time plus accesses to $O(1)$ entries. These auxiliary indices occupy $O(nh\lg\lg n) = O(n\lg\sigma)$ bits of space over all levels.

As described in He et al.'s work [38, Algorithm 2], x and z can be found using a constant number of predecessor and successor queries. Here we only describe how to compute x; the computation of z is similar. First we determine, in tree T, the lowest common ancestor, u', of u and the predecessor of u in $S_{a,b}$. If the weight of u' is in $[a..b]$, then, as illustrated in Fig. 13a, x corresponds to u' and can be determined by the index of u' in $S_{a,b}$. Otherwise, as illustrated in Fig. 13b, we find the successor of u' in $S_{a,b}$ and let the node be v'. The parent of the node in T_k that corresponds to v' will be x.

Summarizing the discussion, the second approach uses $O(\lg\lg n)$ time plus $O(1)$ calls to the ball-inheritance problem. By Lemma 6.3, this requires $O(\lg\lg n + \mathtt{t}(\sigma))$ time. Combining these two approaches, the final time cost is $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$. □

After determining x and z, we start to report nodes. The query range $[p..q]$ must span more than one child range of $[a..b]$; otherwise $[a..b]$ would not be the lowest range that covers $[p..q]$. Let the child ranges of $[a..b]$ be $[a_1..b_1], [a_2..b_2], \ldots, [a_f..b_f]$,

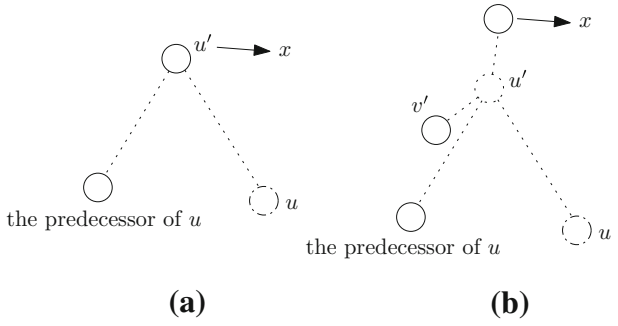

Fig. 13 An illustration for the proof of Lemma 6.4. *Normal* and *dotted circles* represent nodes whose weights are in and not in $[a..b]$, respectively. Node u, which could have a weight in $[a..b]$ or not, is represented by a *dash dotted circle*. **a** The case in which the weight of u' is in $[a..b]$. **b** The case in which the weight of u' is not in $[a..b]$

which are listed in increasing order of left endpoints. Since all but possibly the last of these child ranges are of equal sizes, we can determine in constant time the values of α and β, such that $1 \le \alpha \le \beta \le f$, $[a_\alpha..b_\beta]$ covers $[p..q]$, and $\beta - \alpha$ is minimized. The query range can thus be decomposed into three subranges $[p..b_\alpha]$, $[a_{\alpha+1}..b_{\beta-1}]$ and $[a_\beta..q]$.

The support for the second subrange has been described in [36, Theorem 2]: We first call `node_summarize`(T, x, z) and let the result be $\pi[1..f]$. If $\pi[\gamma] = 1$ for $\gamma \in [\alpha + 1..\beta - 1]$, then we find all nodes whose labels are γ on the path from x to but excluding z by calling $\texttt{lowest\_anc}_\gamma$ repeatedly. Note that each node in the output can be reported by either applying Lemma 6.3 repeatedly or Lemma 6.2 once, and each 1-bit in $\pi[\alpha + 1..\beta - 1]$ can be located in constant time using table lookup. Thus it requires $O(\min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time to report a node.

The remaining part is to support the third subrange in the following lemma; the support for the first subrange is similar.

Lemma 6.5 *The nodes in $A_{u,t} \cap R_{a_\beta,q}$ can be reported using $O(n \lg\sigma)$ bits of additional space and $O((|A_{u,t} \cap R_{a_\beta,q}| + 1) \cdot \min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time.*

Proof We index all T_ℓ's using the first variant of Theorem 1.1 with $m = O(n \lg\lg n)$, for which the weight of a node in T_ℓ is defined to be the weight of its corresponding node in T. Each T_ℓ can be indexed using $O(n \lg\lg n)$ bits of additional space. Hence, these auxiliary data structures occupy $(h - 1) \times O(n \lg\lg n) = O(\lg\sigma/\lg\lg n) \times O(n \lg\lg n) = O(n \lg\sigma)$ bits of additional space in total. Path minimum queries over any T_ℓ can be answered with accesses to the weights of $O(1)$ nodes. This requires $O(\texttt{t}(\sigma))$ time if we use Lemma 6.3, or $O(\lg\sigma/\lg\lg n + 1)$ time if we traverse the conceptual range tree level by level using Lemma 6.2.

Range $[a_\beta..b_\beta]$ is at the $(k + 1)$-st level. Let x' and z' be the nodes in T_{k+1} that correspond to $anc_{a_\beta,b_\beta}(T, u)$ and $anc_{a_\beta,b_\beta}(T, t)$, respectively, as illustrated in Fig. 14. The nodes x' and z' can be computed from x and z in constant time using $\texttt{lowest\_anc}_\beta$ operations and Lemma 6.2. The nodes in T_{k+1} that correspond to the nodes in $A_{u,t} \cap R_{a_\beta,q}$ must locate on the path from x' to and excluding z'. We make use of path minimum queries to find the node, y', with the minimum weight on this

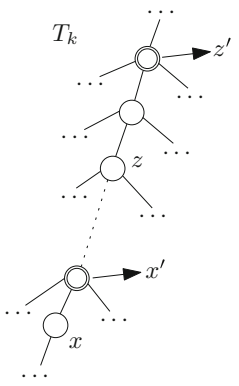

Fig. 14 The root-to-leaf in T_k that goes through both x and z, where nodes with label β are represented as *double circles*. The nodes in T_k that correspond to x' and z' can be determined using $\texttt{lowest\_anc}_\beta$ operations

path. This procedure is terminated if the weight of y' is larger than q; otherwise, the node in T that corresponds to y' is reported, and we recurse on two subpaths obtained by splitting the original path at y'. As we perform a path minimum query for each node reported, the query time is $O((|A_{u,t} \cap R_{a_\beta,q}| + 1) \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$. □

Now we summarize this section in the following theorem.

Theorem 6.6 *An ordinal tree on n nodes whose weights are drawn from a set of σ distinct weights can be represented using $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of space, such that path reporting queries can be supported in $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^\epsilon \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^\epsilon \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

Proof By Lemmas 6.4 and 6.5, it requires $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + (|A_{u,t} \cap R_{p,q}| + 1) \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time to report the nodes in $A_{u,t} \cap R_{p,q}$. The time cost for $A_{v,t} \cap R_{p,q}$ is similar. $\{t\} \cap R_{p,q}$ can be computed in constant time. Summing up these terms, the query time is $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where occ is the size of output. Due to Lemma 6.3, the overall space cost is $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits. □

The data structures developed in Theorem 6.6 match the state of the art of two-dimensional orthogonal range reporting queries [13] when $\sigma = n$. In Sect. 7, we further refine these data structures for the case in which $\sigma < n$.

7 Further Refinements for Range and Path Reporting

Now we further improve the data structures described in Theorem 6.6. The refined data structures incur lower cost in terms of both space and time. The space cost is $O(n \lg\sigma \cdot \mathtt{s}(\sigma))$ bits instead of $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits, while the query time is $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ instead of $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are defined as in Theorem 6.6.

To illustrate our idea, we first develop refined data structures for the two-dimensional orthogonal range reporting problem [13]. In this problem, a set of n points on an $n \times n$ grid is given, and a query asks for the points in an axis-aligned rectangle. Here we consider a more general version of this problem, for which points are drawn from an $n \times \sigma$ grid, where $\sigma \leq n$.

Theorem 7.1 *A set of n points on an $n \times \sigma$ grid, where $\sigma \leq n$, can be represented using $O(n \lg\sigma \cdot \mathtt{s}(\sigma))$ bits of space, such that range reporting queries can be supported in $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^\epsilon \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^\epsilon \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

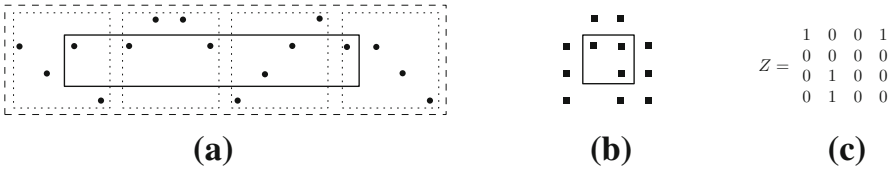

Fig. 15 An illustration for Theorem 7.1. **a** An input point set on a 16×4 grid, which is represented by the *dashed rectangle*. The *dotted rectangles* each represent a subgrid and the *bold rectangle* represents a range query Q. **b** The compressed grid that corresponds to the input point set, where the *bold rectangle* represents the subquery Q_3. **c** The 01-matrix Z that corresponds to the compressed grid

Proof As described in the proof of [6, Lemma 7], we can assume that all the given points have distinct x-coordinates. As illustrated in Fig. 15a, we partition the $n \times \sigma$ grid into $\lceil n/\sigma \rceil$ subgrids, for which the i-th subgrid spans over $[(i-1)\sigma + 1..i\sigma] \times [1..\sigma]$ for $1 \leq i < \lceil n/\sigma \rceil$, and the last one spans over $[(\lceil n/\sigma - 1 \rceil \sigma + 1)..n] \times [1..\sigma]$. Thus each subgrid except the last contains σ input points, and the last one contains at most σ points.

Let $Q = [x_1..x_2] \times [y_1..y_2]$ be the given query. Thus Q spans over the α-th to the β-th subgrids, where $\alpha = \lceil x_1/\sigma \rceil$ and $\beta = \lceil x_2/\sigma \rceil$. We only consider the cases in which $\alpha < \beta$; the other cases can be handled similarly. Q can be split into three subqueries: Q_1, the intersection with the α-th subgrid; Q_2, the intersection with the β-th subgrid; and finally Q_3, the intersection with $(\alpha+1)$-st to the $(\beta-1)$-st subgrids. These subqueries are supported as follows.

For each subgrid, we build the data structures of Bose et al. [6] and one variant of Chan et al.'s [13] structures, which have been summarized in Table 1. These data structures use $O(\sigma \lg \sigma + \sigma \lg \sigma \cdot \mathtt{s}(\sigma)) = O(\sigma \lg \sigma \cdot \mathtt{s}(\sigma))$ bits of space for each subgrid, so the overall space cost is $\lceil n/\sigma \rceil \times O(\sigma \lg \sigma \cdot \mathtt{s}(\sigma)) = O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits. In addition, range reporting queries within a subgrid, e.g., Q_1 or Q_2, can be supported using $O(\min\{\lg \lg \sigma + \mathtt{t}(\sigma), \lg \sigma / \lg \lg n + 1\} + occ' \cdot \min\{\mathtt{t}(\sigma), \lg \sigma / \lg \lg n + 1\})$ time, where occ' is the size of answer.

To support Q_3, we transform the input point set from the original grid into a compressed grid of size $\lceil n/\sigma \rceil \times \sigma$, where a hyperpoint corresponds to one or more points in the original point set. More precisely, an input point (x, y), which is contained in the $\lceil x/\sigma \rceil$-th subgrid, is transformed into a *hyperpoint* $(\lceil x/\sigma \rceil, y)$. See Fig. 15b for an illustration. For each hyperpoint in the compressed grid, we explicitly store the input points that correspond to this hyperpoint as a linked list. Since a subgrid consists of at most σ points, the overall space cost of these linked lists is $O(n \lg \sigma)$ bits.

To answer Q_3, we need to find all the hyperpoints contained in $Q_3' = [\alpha + 1..\beta - 1] \times [y_1..y_2]$. As illustrated in Fig. 15c, we construct a 01-matrix $Z[1..\lceil n/\sigma \rceil, 1..\sigma]$ in which $Z[i, j] = 0$ iff there exists a hyperpoint (i, j). We then encode Z using Brodal et al.'s [8] data structure for two-dimensional range minimum queries, which requires only $O(\lceil n/\sigma \rceil \times \sigma) = O(n)$ bits of space and $O(1)$ query time. To determine all the hyperpoints contained in Q_3', we find all 0-entries in $Z[\alpha + 1..\beta - 1, y_1..y_2]$ by repeatedly performing range minimum queries. Initially, we query on $Z[\alpha + 1..\beta - 1, y_1..y_2]$ for any 0-entry. If there exists some entry $Z[i, j] = 0$, then we know that hyperpoint (i, j) is contained in Q_3'. Furthermore, we divide the remaining entries of

$Z[\alpha+1..\beta-1, y_1..y_2]$ into up to 4 disjoint submatrices and query on them recursively. If no such $Z[i, j]$ exists, then the algorithm terminates and we conclude that there is no more hyperpoint in Q'_3. Thus, the hyperpoints contained in Q'_3 can be found using $O(1)$ time per hyperpoint. By traversing the linked lists associated to these hyperpoints, the points contained in Q_3 can be returned in $O(1)$ time per point.

Summarizing the discussion, the overall space cost is $O(n \lg\sigma \cdot \mathtt{s}(\sigma))$ bits, and queries can be answered in $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ is the size of output. □

We finally generalize the approach of Theorem 7.1 into weighted trees and complete the proof for Theorem 1.3. The cases in which $n = O(\sigma^2)$ have already been handled by Theorem 6.6 ($\lg\lg n = O(\lg\lg\sigma)$ in this case). Here we only consider the cases in which $n = \omega(\sigma^2)$.

Like Sect. 3, we make use of Lemma 2.4 on T with $M = \lceil\sigma^2\rceil$. Thus we obtain $O(n/M)$ cover elements, each being a subtree of size at most $2M$. The root node of a cover element is called a *cover root*. It should be noted again that a cover root could be the root of multiple cover elements. For simplicity, we denote by s_i the i-th cover root in preorder of T. We define the following auxiliary operations with respect to cover roots. Here x is assumed to be a cover root.

- `cover_rank`(T, x) the number of cover roots preceding x in preorder of T;
- `cover_select`(T, i) s_i, i.e., the i-th cover root in preorder of T;
- `cover_depth`(T, x) the number of cover roots between x and root of T;
- `cover_anc`(T, x, i) the i-th lowest cover root along the path from x to the root of T.

Lemma 7.2 *Auxiliary operations* `cover_rank`, `cover_select`, `cover_depth` *and* `cover_anc` *can be supported using* $O(1)$ *time and* $O(n(\lg M)/M)$ *bits of additional space.*

Proof We store a bit vector $B[1..n]$ to mark cover roots, for which $B[j] = 1$ if the j-th node in preorder of T is a cover root. By Lemma 2.2, B can be stored in $O(n(\lg M)/M)$ bits of additional space. We can make use of $\mathtt{rank}_\alpha$ and $\mathtt{select}_\alpha$ operations to compute the rank of a cover root and select the i-th cover root in preorder, respectively.

In addition, we extract all cover roots from T using tree extraction. This gives us a single ordinal tree T', since the root of T must be a cover root. T' is represented using Lemma 2.5, for which we do not store any weight in T'. The overall space cost of storing T' is $O(n/M)$ additional bits, since T' consists of $O(n/M)$ nodes. Since tree extraction preserves ancestor-descendant relationship [35,36,38], `cover_depth` and `cover_anc` can be reduced to `depth` and `level_anc` operations in T', respectively. Therefore, they can be supported in constant time. □

To support path reporting queries, we build the data structures of Theorem 6.6 for each cover element, such that queries inside a cover element can be supported using $O(M \lg\sigma \cdot \mathtt{s}(\sigma))$ bits of space and $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ' \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ' is the size of output. The space cost over all cover elements is $O(n/M) \times O(M(\lg\sigma) \cdot \mathtt{s}(\sigma)) = O(n(\lg\sigma) \cdot \mathtt{s}(\sigma))$ bits.

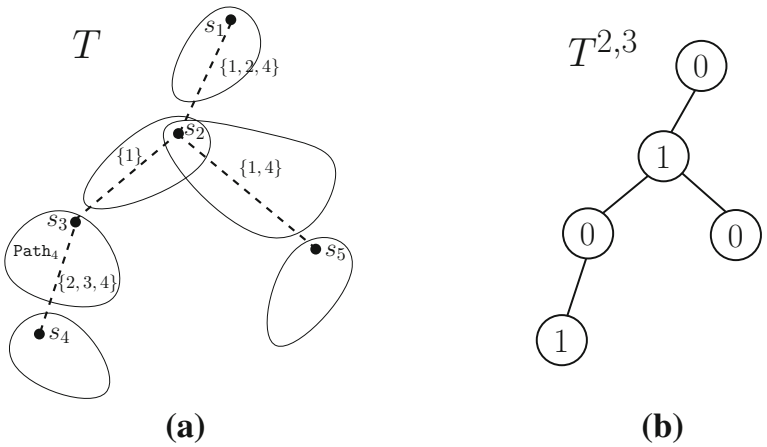

Fig. 16 **a** An input tree T with $\sigma = 4$ and 6 cover elements. The *dashed lines* represent $\texttt{Path}_i$'s, and the numbers alongside each $\texttt{Path}_i$ represent the set of weights on this path. **b** The 01-labeled tree $T^{2,3}$

For each cover root s_i, we denote by $\texttt{Path}_i$ the path from s_i to but excluding the root of the lowest cover element that is an ancestor of s_i, i.e., $\texttt{Path}_i$ contains exactly the nodes in A_{s_i,s_j} for $s_j = \texttt{cover\_anc}(T, s_i, 1)$. In particular, $\texttt{Path}_1$ is empty since s_1 is the root node of T. Note that the length of $\texttt{Path}_i$ is bounded above by $O(\sigma^2)$. We store this path in $O(M \lg \sigma)$ bits of space using the data structures of Alstrup et al. [2], such that given a query range of weights, the nodes along this path whose weights are within this range can be reported in $O(occ' + 1)$ time, where occ' is the number of nodes in the query range. The space cost over all cover elements is $O(n/M) \times O(M \lg \sigma) = O(n \lg \sigma)$ bits.

The last auxiliary data structures are $\sigma(\sigma - 1)/2$ ordinal trees, $T^{p,q}$ for $1 \le p \le q \le \sigma$, all of which are of the same structure as T' (defined in the proof of Lemma 7.2) but on 01-labeled nodes. For each $T^{p,q}$, we assign 1 to a node if the node corresponds to some cover root s_i and $\texttt{Path}_i$ has a node whose weight is in $[p..q]$; otherwise we assign 0 to this node. See Fig. 16 for an example. We maintain these 01-labeled trees using the data structures described in Lemma 6.1, such that the lowest ancestor whose label is 1 of a given node (i.e., $\texttt{lowest\_anc}_1$) can be found in constant time. Each $T^{p,q}$ requires $O(n/M)$ bits of space, and thus the overall space cost for storing all these labeled trees is $O(\sigma^2) \times O(n/M) = O(n)$ bits.

Finally we consider how to answer a given query. Let u and v denote the endpoints of the query path, and let $[p..q]$ denote the query range. As in Theorem 6.6, we only consider the support for $A_{u,t} \cap R_{p,q}$, where t is the lowest common ancestor of u and v, and $A_{u,t}$ is the set of nodes on the path from u to and excluding t.

Let s_a and s_b be the lowest and the highest cover root on the path from u to and excluding t, respectively. The following lemma shows how to compute them.

Lemma 7.3 *The nodes s_a and s_b can be computed in $O(1)$ time.*

Proof The node s_a, which is the lowest cover root on the path from u to t, must be the root of the cover element that contains u if it exists. Using $\texttt{cover\_rank}$ and

$\texttt{cover\_select}$, we can locate s_a in constant time. To compute the highest cover root s_b on the path, we first locate s_c, the root of the cover element that contains t. The highest cover root can be expressed as $s_b = \texttt{cover\_anc}(T, s_a, i)$ for $i = \texttt{cover\_depth}(T, s_a) - \texttt{cover\_depth}(T, s_c) - 1$. □

To answer the query, we only consider the case in which $s_a \neq s_b$; the other cases can be handled similarly. Thus $A_{u,t}$ can be decomposed into A_{u,s_a}, A_{s_a,s_b}, and $A_{s_b,t}$. Here A_{u,s_a} is contained in the cover element rooted at s_a, and $A_{s_b,t}$ except node s_b is contained the cover element rooted at s_b. The nodes along these two subpaths whose weights are in $[p..q]$ can be reported using the data structures described in Theorem 6.6, which have already been stored for each cover element. The query time is $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + (A_{u,s_a} \cup A_{s_b,t}) \cap R_{p,q} \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$.

The subpath from s_a to but excluding s_b can be handled by $T^{p,q}$. Using $\texttt{lowest\_anc}_1$ operations, we can find all cover roots s_i between s_a and s_b such that $\texttt{Path}_i$ has a node whose weight is between p and q, using constant time per s_i. Then, for each of these cover roots s_i, we report all the nodes whose weights are between p and q on $\texttt{Path}_i$, using constant time per node. This is supported by the data structures of Alstrup et al. [2], which have been stored for each $\texttt{Path}_i$.

In sum, the overall space cost is $O(n \lg\sigma \cdot \mathtt{s}(\sigma))$ bits, and the query time is $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where occ is the size of output.

8 Open Problems

We end this article with two open problems.

8.1 Dynamic Data Structures for Path Reporting

One can consider path queries on dynamic trees, for which weights on nodes can be modified and nodes can be inserted or deleted. Several dynamic data structures for path minimum queries have been presented [3,9,39], but much less has been done for dynamic path reporting. In a recent work, He et al. [37] has proposed a linear-space data structure that supports path reporting queries in $O((\lg n/\lg\lg n)^2 + occ \cdot \lg n/\lg\lg n)$ time, where occ is the output size, and the insertion and deletion of a node of an arbitrary degree in $O(\lg^{2+\epsilon} n)$ amortized time, for any constant $\epsilon \in (0, 1)$. This structure is the first and by far the only non-trivial solution to dynamic path reporting. Here we ask whether this data structure can be further improved. As an example, is it possible to reduce the update time to $O((\lg n/\lg\lg n)^2)$ while preserving the same query time and space cost? Whether the $(\lg n/\lg\lg n)^2$ term in query time can be decreased without sacrificing update time or space cost is another interesting problem.

8.2 Adaptive Encoding Complexity of Path Minimum Queries

As shown in Lemma 3.1, $\Omega(n \lg n)$ bits are necessary to encode the answers to all possible path minimum queries in the worst cases. However, the family of trees we constructed for this lower bound have $\Theta(n)$ leaves. As another extreme case, this problem becomes the well-known RMQ problem when the input tree is a single path (or has only two leaves), and thus can be encoded in $2n$ bits. It would be interesting to examine the cases in which the number of leaves, n_L, satisfies that $n_L = o(n)$ and $n_L > 2$.

References

1. Alon, N., Schieber, B.: Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, (1987)
2. Alstrup, S., Brodal, G.S., Rauhe, T.: Optimal static range reporting in one dimension. In: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing, STOC 2001, Heraklion, Crete, Greece, 6–8 July 2001, pp. 476–482, (2001)
3. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, 9–15 July 2000, Proceedings, pp. 73–84, (2000)
4. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Trans. Algorithms **7**(4), 52 (2011)
5. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. **337**(1–3), 217–239 (2005)
6. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, 21–23 August 2009. Proceedings, pp. 98–109, (2009)
7. Bringmann, K., Larsen, K.G.: Succinct sampling from discrete distributions. In: Proceedings of the 45th Annual ACM Symposium on Theory of Computing, STOC 2013, Palo Alto, California, USA, 1–4 June 2013, pp. 775–782, (2013)
8. Brodal, G.S., Davoodi, P., Lewenstein, M., Raman, R., Rao, S.S.: Two dimensional range minimum queries and fibonacci lattices. In: Algorithms—ESA 2012—20th Annual European Symposium, Ljubljana, Slovenia, 10–12 September 2012. Proceedings, pp. 217–228, (2012)
9. Brodal, G.S., Davoodi, P., Rao, S.S.: Path minima queries in dynamic weighted trees. In: Algorithms and Data Structures—12th International Symposium, WADS 2011, New York, NY, USA, 15–17 August 2011. Proceedings, pp. 290–301, (2011)
10. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica **63**(4), 815–830 (2012)
11. Buchsbaum, A.L., Georgiadis, L., Kaplan, H., Rogers, A., Tarjan, R.E., Westbrook, J.: Linear-time algorithms for dominators and other path-evaluation problems. SIAM J. Comput. **38**(4), 1533–1573 (2008)
12. Chan, T.M., He, M., Munro, J.I., Zhou, G.: Succinct indices for path minimum, with applications to path reporting. In: Algorithms—ESA 2014—22th Annual European Symposium, Wroclaw, Poland, 8–10 September 2014. Proceedings, pp. 247–259, (2014)
13. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proceedings of the 27th ACM Symposium on Computational Geometry, SoCG 2011, Paris, France, 13–15 June 2011, pp. 1–10, (2011)
14. Chazelle, B.: Computing on a free tree via complexity-preserving mappings. Algorithmica **2**(1), 337–361 (1987)
15. Chazelle, B., Rosenberg, B.: The complexity of computing partial sums off-line. Int. J. Comput. Geom. Appl. **1**(1), 33–45 (1991)
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)

17. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica **68**(3), 610–625 (2014)
18. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. J. Algorithms **48**(1), 2–15 (2003)
19. Dixon, B., Rauch, M., Tarjan, R.E.: Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM J. Comput. **21**(6), 1184–1192 (1992)
20. Durocher, S., Shah, R., Skala, M., Thankachan, S.V.: Linear-space data structures for range frequency queries on arrays and trees. In: Mathematical Foundations of Computer Science 2013—38th International Symposium, MFCS 2013, Klosterneuburg, Austria, 26–30 August 2013. Proceedings, pp. 325–336, (2013)
21. Durocher, S., Shah, R., Skala, M., Thankachan, S.V.: Top-k color queries on tree paths. In: String Processing and Information Retrieval—20th International Symposium, SPIRE 2013, Jerusalem, Israel, 7–9 October 2013, Proceedings, pp. 109–115, (2013)
22. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica **68**(1), 16–40 (2014)
23. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, 5–12 July 2009, Proceedings, Part I, pp. 451–462, (2009)
24. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms **3**(2), 20 (2007)
25. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
26. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput. **14**(4), 781–798 (1985)
27. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. SIAM J. Comput. **26**(2), 484–538 (1997)
28. Frederickson, G.N.: A data structure for dynamically maintaining rooted trees. J. Algorithms **24**(1), 37–65 (1997)
29. Gabow, H.N.: Data structures for weighted matching and nearest common ancestors with linking. In: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990, San Francisco, California, USA, 22–24 January 1990, pp. 434–443, (1990)
30. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Trans. Algorithms **2**(4), 510–534 (2006)
31. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comput. Sci. **387**(3), 348–359 (2007)
32. Grossi, R., Orlandi, A., Raman, R., Rao, S.S.: More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In: Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science, volume 25 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 517–528, Dagstuhl, Germany, (2009). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik
33. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
34. He, M., Munro, J.I., Satti, S.R.: Succinct ordinal trees based on tree covering. ACM Trans. Algorithms **8**(4), 42 (2012)
35. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: Algorithms and Computation—22nd International Symposium, ISAAC 2011, Yokohama, Japan, 5–8 December 2011. Proceedings, pp. 140–149, (2011)
36. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In Algorithms—ESA 2012—20th Annual European Symposium, Ljubljana, Slovenia, 10–12 September 2012. Proceedings, pp. 575–586, (2012)
37. He, M., Munro, J.I., Zhou, G.: Dynamic path counting and reporting in linear space. In: Algorithms and Computation—25th International Symposium, ISAAC 2014, Jeonju, Korea, 15–17 December 2014, Proceedings, pp. 565–577, (2014)
38. Meng, H., Munro, J.I., Zhou, G.: A framework for succinct labeled ordinal trees over large alphabets. Algorithmica **70**(4), 696–717 (2014)
39. Kaplan, H., S., Nira: Path minima in incremental unrooted trees. In: Algorithms—ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, 15–17 September 2008. Proceedings, pp. 565–576, (2008)

40. King, V.: A simpler minimum spanning tree verification algorithm. Algorithmica **18**(2), 263–270 (1997)
41. Komlós, J.: Linear verification for spanning trees. Combinatorica **5**(1), 57–65 (1985)
42. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nord. J. Comput. **12**(1), 1–17 (2005)
43. Miltersen, P.B.: Lower bounds on the size of selection and rank indexes. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, 23–25 January 2005, pp. 11–12, (2005)
44. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
45. Nivasch, G.: Inverse ackermann without pain. http://www.gabrielnivasch.org/fun/inverse-ackermann
46. Patil, M., Shah, R., Thankachan, S.V.: Succinct representations of weighted trees supporting path queries. J. Discrete Algorithms **17**, 103–108 (2012)
47. Pettie, S.: An inverse-ackermann type lower bound for online minimum spanning tree verification. Combinatorica **26**(2), 207–230 (2006)
48. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43 (2007)
49. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, 22–26 January 2006, pp. 1230–1239, (2006)
50. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
51. Vuillemin, J.: A unifying look at data structures. Commun. ACM **23**(4), 229–239 (1980)
52. Yao, A.C.-C.: Space-time tradeoff for answering range queries (extended abstract). In: Proceedings of the 14th Annual ACM Symposium on Theory of Computing, STOC 1982, San Francisco, California, USA, 5–7 May 1982, pp. 128–136, (1982)