Succinct Dynamic One-Dimensional Point Reporting

Hicham El-Zein
Cheriton School of Computer Science, University of Waterloo, Ontario, Canada N2L 3G1
helzein@uwaterloo.ca

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Ontario, Canada N2L 3G1
imunro@uwaterloo.ca

Yakov Nekrich
Cheriton School of Computer Science, University of Waterloo, Ontario, Canada N2L 3G1
ynekrich@uwaterloo.ca

Abstract

In this paper we present a succinct data structure for the dynamic one-dimensional range reporting problem. Given an interval \([a, b]\) for some \(a, b \in [m]\), the range reporting query on an integer set \(S \subseteq [m]\) asks for all points in \(S \cap [a, b]\). We describe a data structure that answers reporting queries in optimal \(O(k + 1)\) time, where \(k\) is the number of points in the answer, and supports updates in \(O(\log^2 m)\) expected time. Our data structure uses \(B(n, m) + o(B(n, m))\) bits where \(B(n, m)\) is the minimum number of bits required to represent a set of size \(n\) from a universe of \(m\) elements. This is the first dynamic data structure for this problem that uses succinct space and achieves optimal query time.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Succinct Data Structures, Range Searching, Computational Geometry

Digital Object Identifier 10.4230/LIPIcs.SWAT.2018.17

1 Introduction and Motivation

This paper studies the dynamic one-dimensional range reporting problem where the goal is to maintain (under insertion and deletion) a set of integers \(S\) from a universe of size \(m\) to answer range reporting queries efficiently: Given an interval \([a, b]\) for some \(a, b \in [m]\), report all points in \(S \cap [a, b]\). We note that the reporting query is equivalent to the query \(\text{FindAny}(a, b)\) which asks for an arbitrary point \(c\) in \(S \cap [a, b]\): if the interval \([a, b]\) is not empty, we can recurse on \([a, c - 1]\) and \([c + 1, b]\) after obtaining any \(c \in S \cap [a, b]\).

We study this problem in the succinct scenario. In the succinct setting the emphasis is on the space efficiency of the data structure. The goal is to design data structures that occupy optimal or almost-optimal space and at the same time achieve an efficient query cost. This area of research is of interest in theory and practice and is motivated by the need to store a large amount of data using the smallest space possible. In recent years there has been a surge of interest in succinct data structures for computational geometry \([4, 2, 5, 10]\). We refer the reader to the survey by Munro and Rao \([11]\) and the book of Navarro \([17]\) for a more in-depth coverage of succinct data structures.

1 This work was sponsored by the NSERC of Canada and the Canada Research Chairs Program.
Related Work. One-dimensional range reporting is a well studied problem. Miltersen et al. [13] presented a data structure for the static version of this problem that uses $O(n \lg m)$ words and answers queries in constant time per reported element. Alstrup et al. [1] later presented an improved data structure with the same query time that uses $O(n)$ words, i.e., $O(n \lg m)$ bits. Goswami et al. [7] presented a succinct data structure that further improved the space usage to $B(n, m) + o(B(n, m))$ bits while preserving the query time where $B(n, m) \approx n \lg (m/n)$ is the minimum number of bits required to represent a set of size $n$ from a universe of $m$ elements.

For the dynamic version of this problem Mortensen et al. [14] presented a data structure that uses a linear number of words and answers queries in $O(t_q)$ time per reported element. Alstrup et al. [1] later presented an improved data structure with the same query time that uses $O(n \lg m)$ time and queries in constant time. In Section 4 we present a fully-dynamic succinct one dimensional point reporting data structure that supports updates in expected $O(\lg^2 m)$ time using $o(n)$ workspace. We defer the details of this construction to the end in Section 5 due to its technical nature.

Our Results. We start with some preliminaries in Section 2. In Section 3 we present a semi-dynamic succinct range reporting data structure that supports deletions in expected $O(\lg^2 m)$ time and queries in constant time. In Section 4 we present a fully-dynamic succinct range reporting data structure that supports updates in expected $O(\lg^2 m)$ time and queries in constant time. Our results depend on the ability to construct a static succinct one dimensional point reporting structure in $O(n \lg^2 m)$ time using $o(n)$ workspace. We defer the details of this construction to the end in Section 5 due to its technical nature.

2 Preliminaries

In this section we review some previous results that will be used in the rest of this paper.

2.1 One-Dimensional Point Reporting

First we review the data structure of Alstrup et al. [1] for static one-dimensional range reporting. We start by defining some notations. Let $x \oplus y$ denote the binary exclusive-or of $x$ and $y$. Given a $w$-bit integer $x$ let $x \downarrow i = x/2^i$ denote the rightmost $w$ bits of the result of shifting $x$ $i$ bits to the right. Similarly let $x \uparrow i = x \cdot 2^i \mod 2^w$ denote the rightmost $w$ bits of the result of shifting $x$ $i$ bits to the left. Finally, denote by $\text{msb}(x)$ the position of the most significant bit (or leftmost one bit) of $x$.

Given a set of integers $S$ the goal is to store $S$ while supporting the query $\text{FindAny}(a, b)$ which returns an element in $S \cap [a, b]$. Denote by $T$ the classic binary tree with $2^w$ leaves where all leaves have depth $w$. The leaves are numbered $0, \ldots, 2^w - 1$ from left to right while the internal nodes are labeled in a manner similar to an implicit binary heap. The root is the first node, and the children of a node $v$ are $2v$ and $2v + 1$. As noted in [1] the $d$th ancestor of $v$ is $v \downarrow d$ and the lowest common ancestor of two leaves $a$ and $b$ is the $(1 + \text{msb}(a \oplus b))$th ancestor of $a$ or $b$. Thus the lowest common ancestor of two leaves can be computed in constant time.

Given a node $v \in T$ let $\text{left}(v)$ and $\text{right}(v)$ denote the left and right children of $v$, and let $S_v$ denote the subset of $S$ that is in the subtree rooted at $v$. A node $v$ is branching if both $S_{\text{left}(v)}$ and $S_{\text{right}(v)}$ are not empty. To answer a query $\text{FindAny}(a, b)$ it is sufficient to compute the lowest common ancestor $v$ of $a$ and $b$; when $v$ is computed, either $\max S_{\text{left}(v)}$ or $\min S_{\text{right}(v)}$ is in $[a, b]$, or $[a, b]$ is empty. Thus by storing the values max $S_{\text{left}(v)}$ and
Theorem 1 ([12]). There is an \((m, n, O(n))\)-family of perfect hash functions \(H\) such that any hash function \(h \in H\) can be represented in \(\Theta(n \lg \lg n)\) bits and evaluated in constant time for \(m \leq 2^n\). The perfect hash function can be constructed in expected \(O(n)\) time.

As noted in [1] a corollary of the previous theorem is the following.

Corollary 2. A sparse array of size \(m \geq n\) with \(n\) initialized entries that contain \(b = \Omega(\lg \lg n)\) bits each can be stored using \(O(nb)\) bits, so that any initialized entry can be accessed in \(O(1)\) time. The expected preprocessing time of this data structure is \(O(n)\).

3 Semi-Dynamic Succinct One-Dimensional Point Reporting

Although Goswami et al. [7] presented a succinct data structure for one-dimensional range reporting, it is not clear what is the construction time of their data structure. In Section 5 we utilize succinct data structure techniques to improve the data structure in [1] so that it
uses $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits and can be constructed in $O(n \lg^\varepsilon m)$ time using $o(n)$ extra bits of space. The details are deferred to Section 5 due to their technical nature.

**Theorem 3.** There exists a succinct $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$-bit data structure that supports one-dimensional range reporting queries in $O(k + 1)$ time where $k$ is the number of points within the query. Additionally given the point set in sorted order, this data structure can be constructed in expected $O(n \lg^\varepsilon m)$ time using $o(n)$-bits workspace.

The data structure for one-dimensional range reporting can be dynamized so that queries are supported in deterministic $O(k)$ time and updates in expected $O(\lg^\varepsilon m)$ time while the space usage is $O(n)$ words [14]. Our aim is to reduce the space to the information theoretic lower bound plus a lower order term. In this section we present a semi-dynamic succinct one-dimensional range reporting data structure that supports queries and deletions but does not support insertions.

**Data Structure.** We store the data structure from Theorem 3 and call it $P$. We divide the points into blocks of size $\lg^2 m$ and we store predecessor and successor data structures that can answer queries in each block independently using $o(\mathcal{B}(n, m))$ bits as described in [4]. We also store a dynamic data structure [14] $D$ on the endpoints of each block. Furthermore, each block is divided into subblocks of size $\lg n/2$ and stores a dynamic data structure [14] $D_i$ ($1 \leq i \leq n/\lg^2 m$) on the ranks (within the block) of the endpoints of each subblock. We also store a compressed bit vector ([8], Theorem 2) $B$ of size $n$ that indicates which points were deleted. Finally, we store a lookup table $T$ that can report for any range the $0$ bits in a bit vector of size $\lg n/2$.

**Query.** To report the points within an interval $[a, b]$ we query $D$ on the interval. Then for each point reported with rank $k$ we query the $\lfloor (k/2) \rfloor$th and $\lceil (k/2) \rceil$th blocks.

To query the $k^{th}$ block we first reduce the problem to the rank space by finding the rank of the successor of $a$ and the predecessor of $b$ within the block. Next, we query $D_k$ for the non-empty subblocks within the block and use $T$ to report the points in the subblock.

If the query to $D$ does not return any point then either $[a, b]$ is empty or $[a, b]$ is contained fully within a block. To determine which block contains $[a, b]$ we query $P$ to get the rank of a random point in $[a, b]$ from that we determine which block contains $[a, b]$. Afterwards we proceed within the block as described above.

**Deletions.** To delete a point $p$ we first query to check that the interval $[p, p]$ is not empty. We obtain the rank $k$ of $p$ by querying $P$, and then we set the $k^{th}$ bit in $T$ to $1$. Now we know that the point $p$ is in the $s = 2(k \mod \lg^2 m)/\lg n$th subblock of the $b = (k/\lg^2 m)$th block. We check if the $s^{th}$ subblock is empty. If that is so we remove its endpoints from $D_{(k/\lg^2 m)}$. Then we check if the $b^{th}$ block is empty. In that case we remove its endpoints from $D$. The expected running time is $O(\lg^\varepsilon m)$.

**Space Analysis.** $P$ uses $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits and $D$ contains $O(n/\lg^2 m)$ points thus uses $O(n/\lg m)$ bits. Each $D_i$ ($1 \leq i \leq n/\lg^2 m$) contains $O(\lg^2 m/\lg n)$ points from a universe if size $\lg^2 m$ thus uses $O(\lg^2 m \lg \lg m/\lg n)$ bits. The $D_i$ structures use $O(n \lg \lg m/\lg n)$ bits in total. If $\lg \lg m \notin o(\lg n)$ then $n < \lg^c m$ for some constant $c$. In that case we use a slightly different approach. We reduce the problem to the rank space from the beginning to make the universe size $n$, so $D$ uses $O(n/\lg n)$ bits and the $D_i$ structures use $O(n \lg \lg n/\lg n)$ bits in total. The table $T$ uses $O(\sqrt{n} \lg^2 n \lg \lg n)$ bits and finally the compressed bit vector uses $o(n)$ as long as the number of deletions is $o(n)$. In total the space remains $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits.
Construction Time and Workspace. $P$ can be constructed in expected $O(n \log^2 m)$ time using $o(n)$ extra bits of space. $D$ can be constructed in expected $O(n/\log^2 m)$ time using $O(1)$ extra words of space. Each $D_i$ can be constructed in expected $O((\log^2 m/\log n) \log^2 m)$ time using $O(1)$ extra words of space, so all the $D_i$’s can be constructed in expected $O((n/\log n) \log^2 m)$ time using $O(1)$ extra words of space. $T$ can be constructed in $o(n)$ time using $o(n)$ extra bits of space. In total the construction time and workspace are dominated by the cost of constructing $P$ and remain the same as in Theorem 3.

**Theorem 4.** There exists a semi-dynamic succinct $B(n, m) + o(B(n, m))$-bit data structure that supports one-dimensional range reporting queries in $O(k + 1)$ time where $k$ is the number of points within the query, and point deletions in expected $O(\log^2 m)$ time as long as the number of deletions is $o(n)$. Additionally given the point set in sorted order, this data structure can be constructed in expected $O(n \log^2 m)$ time using $o(n)$-bits workspace.

4 Fully-Dynamic Succinct One-Dimensional Point Reporting

4.1 Fully-Dynamic Structure with Amortized Updates

We first present a fully dynamic solution that uses $B(n, m) + o(B(n, m))$ bits of space and supports queries in $O(k)$ time and updates in amortized expected $O(\log^2 m)$ time.

We divide the universe of size $m$ into $n/\log^2 m$ chunks of equal size and maintain a fully dynamic [14] data structure $B$ to keep track of the nonempty chunks. $B$ is maintained throughout the data structure updates. Whenever a point is inserted we insert both endpoints of its chunk into $B$. Moreover whenever a chunk becomes empty we remove its endpoints from $B$. For each chunk $b_i$ ($1 \leq i \leq n/\log^2 m$) we maintain two data structures: $S_i$ and $D_i$. $S_i$ is the compressed semi-dynamic range reporting structure described in Theorem 4 and $D_i$ is the fully dynamic data structure described in [14]. We maintain the invariant that size($D_i$) < size($S_i$)/$\log^2 n$ for all $i$ where $n = \sum_i$ size($S_i$). Once size($D_i$) = size($S_i$)/$\log^2 n$ we rebuild $S_i$ and merge $D_i$ with it. The time needed to rebuild $S_i$ will be $O($size($S_i$)$\log^2 m$ which we can charge to the elements inserted into $D_i$ at a cost of $O(\log^2 m)$ per element. Moreover if the total number of elements increase by a constant factor or if $n/\log^2 n$ elements were deleted from the collections $S_i$ we rebuild the whole data structure. The time needed to rebuild the whole structure is $O(n \log^2 m)$ and will be charged to the new elements inserted if the size doubles at a cost of $O(\log^2 m)$ per element, or to the elements deleted at a cost of $O(\log^2 m)$ per element.

To report all the points within an interval $[a, b]$ we query $B$ to get the non-empty chunks. Whenever a non-empty chunk $i$ is reported we query both $S_i$ and $D_i$. If $[a, b]$ is completely within one chunk we get its index $i = \lfloor b \log^2 m/n \rfloor$, and then we query $S_i$ and $D_i$.

The space used by $B$ is at most $O(n/\log m)$ bits. and the space used by all the $D_i$ structures is:

$O(n \log (m \log^2 m/n)/\log^2 n) = O((n \log (m/n)/\log^2 n) + (n \log \log n/\log^2 n))$

$= o(B(n, m)).$

The space used by all the structures $S_i$ is $B(n, m) + o(B(n, m))$ bits. In total the space used is $B(n, m) + o(B(n, m))$ bits.

**Theorem 5.** There exist a dynamic succinct $B(n, m) + o(B(n, m))$-bit data structure that supports one-dimensional range reporting queries in $O(k + 1)$ time where $k$ is the number of points within the query, and updates in amortized expected $O(\log^2 m)$ time.
4.2 Fully-Dynamic Structure with Worst Case Updates

Next, we present a fully-dynamic succinct one-Dimensional range reporting structure that supports queries in $O(k)$ time and insertions and deletions in expected $O(\log^* m)$ time. Our data structure uses techniques similar to the ones presented in [9, 15, 16].

**Data Structure.** We define a parameter $n_f = \Theta(n)$; the value of $n_f$ changes as $n$ becomes too large or too small. We divide $m$ into $(n_f / \log^2 n_f)$ chunks each of size $((m \log^2 n_f) / n_f)$ and we store a dynamic range reporting structure $B$ with a universe of size $2(n_f / \log^2 n_f)$ on the endpoints of the non-empty chunks. For each chunk $b$ where $1 \leq b \leq (n_f / \log^2 n_f)$ we store the following:

- $k^b_f$ an estimate of $k$ the number of points in the chunk. $k^b_f = \Theta(k)$, the value of $k^b_f$ changes as $k$ becomes too large or too small.

**Data Structures** $C^b_1, \ldots, C^b_{\log^* n_f}$. These structures are the succinct semi-dynamic structures described in the previous section. They partition the chunk into sub-chunks of possibly different sizes, each containing $\Theta(k^b_f / \log^* n_f)$ points.

**Data Structures** $D^b_1, \ldots, D^b_{\log^* n_f}$. These structures are the fully dynamic structures described in [14].

- $F^b$ a fusion tree on the endpoints of the $C^b_i$ data structures.

Queries are answered in a manner similar to the previous subsection. To report all the points within an interval $[a, b]$ we query $B$ to get the non-empty chunks. Whenever a non-empty chunk (say the $b^{th}$ chunk) is reported we query $F^b$ to get the sub-chunks it spans. For each sub-chunk (say the $s^{th}$ sub-chunk) we query both $C^b_s$ and $D^b_s$.

**Insertions.** To insert the new point $p$ we compute the chunk $b = \lceil (p \log^2 n_f) / n_f \rceil$ that $p$ belongs to. If the $b^{th}$ chunk is empty we insert its endpoints into $B$. Next, we check if any structure in the $C^b$ collection is being rebuilt. In that case we spend $\Theta(\log^* n_f)$ time rebuilding it. Then we determine the $s^{th}$ sub-chunk that $p$ belongs to using $F^b$. Finally, we insert $p$ into $D^b_s$.

In each chunk we run the following background process. After each series of $\delta = k^b_f / (\log^* n_f \log n_f)$ insertions we identify the $s^{th}$ sub-chunk with the largest number of inserted points and rebuild $C^b_s$ during the next $\delta$ updates in that chunk. The re-building works as follows. We construct a semi-dynamic data structure $\overline{C}^b_s = C^b_s \cup D^b_s$. If a point is inserted into this sub-chunk, we store it in the additional data structure $\overline{D}^b$. When $\overline{C}^b_s$ is completed we set $C^b_s := \overline{C}^b_s$ and $D^b_s := \overline{D}^b$. Thus at any time only one sub-chunk of a chunk is re-built. This method guarantees that the number of inserted elements into $D^b_s$ does not exceed $k^b_f / \log^* n$ as follows from a Theorem of Dietz and Sleator:

**Lemma 6 ([3], Theorem 5).** Suppose that $x_1, \ldots, x_g$ are variables that are initially zero. Suppose that the following two steps are iterated:

1. we add a non-negative real value $a_i$ to each $x_i$ such that $\sum a_i = 1$
2. set the largest $x_i$ to 0.

Then at any time $x_i \leq 1 + h_{i-1}$ for all $i$, $1 \leq i \leq g$, where $h_i$ denotes the $i$-th harmonic number.

Let $m_s$ be the number of inserted elements into $D^b_s$ and $x_s = m_s / \delta$. Every iteration of the background process sets the largest $x_s$ to 0 and during each iteration $\sum x_s$ increases by 1. Hence the value of $x_s$ can be bounded from above by: $x_s \leq 1 + h_{\log^* n_f}$ for all $s$ at all times.
Thus \( m_s = O((k_f^i / \log^2 n_f) \log \log n_f) = O(k_f^i / \log^2 n_f) \) for all \( i \) because \( h_i = O(\log i) \), and the total size of the \( D^b \) collection is \( O((k_f^i / \log^2 n_f) \log^2 n_f) = O(k_f^i / \log^2 n_f) \).

Once the value of \( k_f^i \) becomes too big or too small we rebuild the whole chunk during the next \( k_f^i / \log^3 n_f \) updates (spending \( O(\log^4 n_f) \) time per update). The old chunk is locked such that only deletions are allowed. We rebuild the chunk with an updated value of \( k_f^i \) and as points are inserted into the new chunk we delete them from the old one to preserve space. If the size of the sub-chunk becomes too big we split it into two and update \( F^b \) accordingly.

**Deletions.** Deletions are similar to insertions. To delete a point \( p \) we compute the chunk \( b = \lfloor (p \log^2 n_f) / n_f \rfloor \) that \( p \) belongs to. Then we check if any structure in the \( C^b \) collection is being rebuilt. In that case we spend \( \Theta(\log^3 n_f) \) time rebuilding it. Next, we determine the sub-chunk \( s \) that \( p \) belongs to using \( F^b \). Finally, we delete \( p \) from \( C^b_s \) and \( D^b_s \).

In each chunk we run a background process similar to the process run for insertions. After each series of \( \delta \) deletions, we identify the \( s^b \) sub-chunk with the largest number of deletions and rebuild \( C^b_s \) during the next \( \delta \) updates in that chunk. This method guarantees that the number of deleted elements in the \( C^b \) collection does not exceed \( k_f^i / \log^2 n_f \). If the size of a sub-chunk becomes too small we merge it with the neighboring sub-chunk and update \( F^b \) accordingly. Moreover if a chunk becomes empty we delete its endpoints from \( B \).

**Space Analysis.** The space used by \( B \) is \( O(n / \log n) \). The space used by all the \( C_i \) structures in all chunks is \( B(n, m) + o(\mathcal{B}(n, m)) \) bits. The total size of all the \( D \) structures is \( O(n_f / \log^2 n_f) \) so they use at most:

\[
O(n \log (m \log^2 n_f) / \log^2 n_f) = O((n \log (m / n) / \log^2 n) + (n \log \log n / \log^2 n))
\]

\[
= o(\mathcal{B}(n, m)).
\]

The space used by the fusion trees in all chunks is:

\[
O(n \log^2 n \log (m \log^2 n_f) / \log^2 n_f) = O((n \log (m / n) / \log^{2-\varepsilon} n) + (n \log \log n / \log^{2-\varepsilon} n))
\]

\[
= o(\mathcal{B}(n, m)).
\]

Thus the total space is \( B(n, m) + o(\mathcal{B}(n, m)) \) bits.

Once the value of \( n_f \) becomes too big or too small, we rebuild the whole data structure in the background during the next \( n_f / \log^3 n_f \) updates (spending \( O(\log^4 n_f) \) time per update). We replace the chunks from left to right. The chunk being replaced is locked such that only deletions are allowed. We rebuild that chunk with an updated value and as points are inserted into the new chunk we delete them from the old one to preserve space.

**Theorem 7.** There exist a dynamic succinct \( \mathcal{B}(n, m) + o(\mathcal{B}(n, m)) \)-bit data structure that supports one-dimensional range reporting queries in \( O(k + 1) \) time where \( k \) is the number of points within the query, and updates in expected \( O(\log^2 m) \) time.

## 5 Succinct Static One-Dimensional Point Reporting With Fast Construction Time

In this section we prove Theorem 3. Denote by \( T \) the classic binary tree with \( 2^w \) leaves where all leaves have depth \( w \) as described in subsection 2.1. Let \( P \) be the set of nodes in \( T \) with non-empty subtrees and \( V \) the set of branching nodes in \( T \) union the leaves of \( T \) and its root. Let \( T_V \) be the tree formed from \( T \) by deleting all vertices in \( T - P \) then contracting
all vertices in $P - V$. Given a node $x \in T_V$ denote by $T(x)$ its corresponding node in $T$, conversely, given a node $x \in V$ denote by $T_V(x)$ its corresponding node in $T_V$. We fix a constant $\varepsilon = 1/k$, and let $H_i = \lg^{(k-i)/k} m$ where $1 \leq i < k$. Finally, given a node $u$ in $T$ we define $\pi_i(u)$ to be the nearest ancestor of $u$ whose depth is a multiple of $H_i$.

**Data Structure.** We store the coordinates of the points in $B(n, m) + o(B(n, m))$ bits. Also we store $T_V$ using $4n + o(n)$ bits using the tree representation of Navarro and Sadakane [18] which allows the following operations in constant time:

- **lmost-leaf(i) / rmost-leaf(i):** given the preorder number of a node return the preorder number of the leftmost(rightmost) leaf of node $i$.
- **leaf-rank(i):** given the preorder number of a leaf $i$ returns the number of leaves to the left of $i$.

In addition we store in $o(n)$ bits the index described in [6] that enables conversion between $\tau$-names of the nodes in $T_V$ and their preorder numbers.

To maintain the mapping between the labels of the branching nodes in $T$ with their preorder numbers in $T_V$ we store the following tables using Corollary 2:

- $M_1$: for each node $x \in V$ with root $(T_V(x)) = T_V(x)$ we store the value $\tau_1(T_V(x))$ in a table $M_1$. Since $T_V$ is a binary tree, it is possible that $T_V(x)$ belongs to two different micro trees $p_0$ and $p_1$. In that case we store both $p(M_0)$ and $p(M_1)$.
- $M_2$: for each node $x \in V$ we store in a table $M_2$ the values $\tau_2(T_V(x))$, $\tau_3(T_V(x))$, and a bit that indicates to which micro tree does $T_V(x)$ belongs to if root $(T_V(x))$ belongs to two different micro trees.
- $M_3$: for each node $x \in V$ we store the distance from $x$ to $T(root(T_V(x)))$ in a table $M_3$.

Finally, given a node $p$ in $P$ we need to compute its nearest branching ancestor. To achieve this we use the same technique as in [1] but with bootstrapping. We store $k - 1$ tables $D_1, \ldots, D_{(k-1)}$ using Corollary 2. $D_1$ contains the distances to the nearest branching ancestor for all nodes $u$ in $P$ satisfying $\pi_1(u) = u$. $D_i (2 \leq i < k - 1)$ contains the distances to the nearest branching ancestor for all nodes $u$ in $P$ satisfying the conditions $\pi_{(i-1)}(u)$ is closer to $u$ than the nearest branching ancestor of $u$ and $\pi_i(u) = u$. Finally, $D_{(k-1)}$ contains the distances to the nearest branching ancestor for all nodes $u$ in $P$ satisfying the conditions: $\pi_{(k-2)}(u)$ is closer to $u$ than the nearest branching ancestor of $u$ and $\pi_{(k-1)}(u) = u$, or $\pi_{(k-1)}(u)$ and $\pi_{(k-2)}(u)$ are closer to $u$ than the nearest branching ancestor of $u$. More formally we define:

- $B_1: B_1(z) = 1$ if $\pi_1(z) = z$ and $\exists u \in V$ such that $\pi_1(u) = z$, otherwise $B_1(z) = 0$.
- $B_i(1 < i < k - 1)$: $B_i(z) = 1$ if $B_{(i-1)}(\pi_{(i-1)}(z)) = 1$, $\pi_i(z) = z$, and $\exists u \in V$ such that $\pi_i(u) = z$, otherwise $B_i(z) = 0$.

and store the following tables using Corollary 2:

- $D_1$: which contain the distance to the nearest branching ancestor for all nodes $u$ in $P$ satisfying $\pi_1(u) = u$.
- $D_i (2 \leq i < k - 1)$: which contain the distance to the nearest branching ancestor for all nodes $u$ in $P$ satisfying: $B_{(i-1)}(\pi_{(i-1)}(u)) = 1$ and $\pi_i(u) = u$.
- $D_{(k-1)}$: which contain the distance to the nearest branching ancestor for all nodes $u$ in $P$ satisfying: $B_{(k-2)}(\pi_{(k-2)}(u)) = 1$ and $(\pi_{(k-1)}(u) = u$ or $B_{(k-1)}(\pi_{(k-1)}(u)) = 1$.

**Query.** Given a query $\text{FindAny}(a, b)$ we first find the nearest common ancestor $p$ of $a$ and $b$. Then we get $k - 1$ candidate nearest branching ancestor $v_1, \ldots, v_{(k-1)}$ of $p$ using $D_1, \ldots, D_{(k-1)}$. Afterwards for each $v_i$ we need to compute the preorder number of $v_i$ in $T_V$. To achieve this goal we get $\tau_2(T_V(v_i))$, $\tau_3(T_V(v_i))$, and the bit $b$ indicating which micro tree $v_i$ belongs to from $M_2$. Next, we compute $u_i = T(root(T_V(v_i)))$ after obtaining its distance
from \(v_i\) using \(M_3\). Afterwards we query \(M_1\) for \(\tau_1(T_V(u_i)) = p(\mu_b)\). After obtaining the 
\(\tau\)-name of \(T_V(v_i)\) we get its preorder number, and then we check the ranks of the leftmost 
and rightmost leaves of \(v_i\)‘s left and right child. If one of them is within \([a, b]\) we return its 
value. If for all \(v_i\) no element was found within \([a, b]\) we return that \(S \cap [a, b]\) is empty.

**Space Analysis.** Storing the points coordinates uses \(B(n, m)\) bits. The tree \(T_V\) uses 
\(4n + o(n)\) bits. The tables \(M_2, M_3\) contain \(O(n)\) entries each of size \(O(\log \log m)\) so they 
use \(O(n \log \log m)\) bits. The table \(M_1\) contains \(O(n/\log n)\) entries each of size \(O(\log n)\) so it 
uses \(O(n)\) bits. The table \(D_1\) contains \(O(n \log m / \log^{(k-1)/k} m) = O(n \log^2 m)\) entries of size 
\(O(\log \log m)\) bits each so it uses \(O(n \log^2 m \log \log m)\) bits. Moreover each table \(D_i\) (\(1 < i < k - 1\)) 
contains \(O(n(H_{i-1}/H_i)) = O(n \log^2 m)\) entries of size \(O(\log \log m)\) bits so they use a 
total of \(O(n \log^2 m \log \log m)\) bits. Finally, we need to bound the size of \(D_{k-1}\). The number of 
entries due to \(\pi_{k-1}(u) = u\) is \(O(n(H_{k-1}/H_k)) = O(n \log^2 m)\). To bound the entries due to 
\(B_{k-1}(\pi_{k-1}(u)) = 1\) notice that the subtree \(T_2\) of height \(H_{k-1}\) rooted at \(z = \pi_{(k-1)}(u)\) 
will contain \(s > 1\) entries, and will have at most \(s + 1 < 2s\) leaves that are nodes in \(P\). Thus it 
will contribute at most \((2H_{k-1}s)\) entries. Since there are at most \(n - 1\) branching nodes 
the total number of entries due to \(B_{(k-1)}(\pi_{(k-1)}(u)) = 1\) is \(2 H_{k-1} \log m = O(n \log^2 m)\). \(D_{k-1}\) 
uses \(O(n \log^2 m \log \log m)\) bits because each entry in \(D_{(k-1)}\) is of size \(O(\log \log m)\) bits. In total 
the space used is \(B(n, m) + O(n) + O(n \log^2 m \log \log m)\) bits.

**Construction Time.** In a manner similar to [1] we can identify \(V\) in \(O(n)\) time, and then 
construct \(T_V\) also in \(O(n)\) time. The tables \(M_1, M_2,\) and \(M_3\) can be constructed in expected 
\(O(n(\log \log m))\) time. Finally, the tables \(B_i\), where \(1 \leq i < k\) can be constructed in expected 
\(O(n \log^2 m)\) time by identifying the \(O(n \log^2 m)\) entries and building the tables. The workspace 
is \(O(n)\) words.

**Reducing Space.** To further reduce the space we use a well known trick and split the 
universe \([n]\) into \(n\) ranges \(r_1, \ldots, r_n\) each of size \(m/n\). We construct a bit vector \(B\) of size \(2n\) 
bits with rank and select queries. \(B\) stores a zero for each range \(r_i\) followed by \(n_i\) ones where 
\(n_i\) is the number of points in the range \(r_i\). To count the number of points before a range \(r_i\) we 
use a select query to get the position of the \(i^{th}\) zero in \(B\), and then use a rank query to count 
the number of ones before that position. We store a separate data structure for each range. 
To locate the data structures for any range \(r_i\) within \(A\) we count the number of points in the 
ranges \(r_j\) for \(j < i\), and then scale that number. Given a query \text{FindAny}(a, b)\ we check if \([a, b]\) 
spans a non-empty range as follows. We use a rank query to get the number of ones \(k\) before 
the \([\lceil an/m \rceil]\) zero. Then we check if the \((k+1)^{th}\) element is within \([a, b]\) and return it in that 
step. Otherwise we query the data structure corresponding to the \(\lceil (an/m) \rceil\) range. The 
total space used is \(B(n, m) + O(n) + O(n(\log (m/n))^2 \log \log (m/n)) = B(n, m) + O(B(n, m))\) 
bits.

If \(O(n)\) is not a lower order term then \(n > m/c\) for some constant \(c\). In that case we adopt 
a different approach and store the points in a compressed bit vector of size \(m\). To answer a 
query \text{FindAny}(a, b)\ we use a rank query to get the number of ones \(k\) before position \(a\), and 
then we use a select query to get the position of the \((k+1)^{th}\) one. If that position is within 
\([a, b]\) we return it otherwise \(S \cap [a, b]\) is empty. The space used is now \(B(n, m) + O(B(n, m))\) 
bits.

**Reducing Construction Workspace.** To further improve the construction workspace we 
divide \(n\) into \(\log^2 m\) ranges each containing \(n/\log^2 m\) points and build a separate data structure 
for each of them. We note that the universe size in each range may vary. Additionally we
store a fusion tree $F$ on the endpoints of each range. Given a query $\text{FindAny}(a, b)$, we check if the successor of $a$ in $F$ is within $[a, b]$ and return it in that case. Otherwise we query the range containing the successor of $a$.

References


