Range Aggregate Maximal Points in the Plane

Ananda Swarup Das[1], Prosenjit Gupta[2], Anil Kishore Kalavagattu[3], Jatin Agarwal[3], Kannan Srinathan[3], and Kishore Kothapalli[3]

[1] Kalinga Institute of Industrial Technology, Bhubaneswar, India
anandaswarup@gmail.com
[2] Heritage Institute of Technology, Kolkata, India
prosenjit_gupta@acm.org
[3] International Institute of Information Technology, Hyderabad, India
{anilkishore,jatin.agarwal}@research.iiit.ac.in,
{srinathan,kkishore}@iiit.ac.in

Abstract. In this work, we study the problem of reporting and counting maximal points in a query rectangle for a set of n integer points that lie on an $n \times n$ grid. A point is said to be maximal inside a query rectangle if it is not dominated by any other point inside the query rectangle. Our model of computation is unit-cost RAM model with word size of $O(\log n)$ bits. For the reporting version of the problem, we present a data structure of size $O(n\frac{\log n}{\log\log n})$ words and support querying in $O(\frac{\log n}{\log\log n} + k)$ time where k is the size of the output. For the counting version, we present a data structure of size $O(n\frac{\log^2 n}{\log\log n})$ words which supports querying in $O(\frac{\log^{\frac{3}{2}} n}{\log\log n})$. Both the data structures are static in nature. The reporting version of the problem has been studied in [1] and [5]. To the best of our knowledge, this is the first sub-logarithmic result for the reporting version and the first work for the counting version of the problem.

1 Introduction

Range Searching is one of the most widely studied topics in computational geometry and is defined as follows: given a set P of geometrical objects (lines, points etc.), preprocess them into a data structure D such that given a query q (rectangle, circle triangle etc.), we can efficiently report the objects in $P \cap q$. However, at times, the business tools using range aggregate queries want to have an "interesting summarization" of the results than the entire result set. In database literature, the interesting points are often discovered by finding the *skyline points* for the data set. In computational geometry, the skyline points are nothing but the set of maximal points for a data set. A point p is said to be dominating point p' if $p_x \geq p'_x$ and $p_y \geq p'_y$. Given a point set R, the set of maximal points R' is the set of points which are not dominated by any other points in the set R. In this work, we study the problems of reporting and counting the number maximal points inside a query rectangle. Our model of computation is word-RAM model with word size of $O(\log n)$ bits.

1.1 Preliminaries

Rank Query: For the efficient solutions of the two problems, we need to use the *rank* query of [2]. Let S be a binary string of zeros and ones. For any character $c \in \{0, 1\}$, and any position i, a rank query denoted by $rank_c(S, i)$ reports the number of c in S from position 1 to position i. For rank query, the following result is known from [2].

Lemma 1. *A binary string $S(1, n)$ can be represented using $n + o(n)$ bits of space while supporting rank query in $O(1)$ time.*

Finding the position of the Least Significant Bit set to 1: Let s be a binary string such that $s \neq 0$. We can find the least significant bit of s set to one by finding the $2's$ complement of s. Call the $2's$ complement of s as s'. Then perform $s \wedge s'$. Say for example $s = 011010$. The least significant bit which is set to 1 is one. Find the $2's$ complement of s. It is $100101 + 1 = 100110$. Perform $s \wedge s' = 000010$. The least significant bit set to one can therefore be found using a look up table storing the $\log n$ values that are powers of two. For more details see [7].

1.2 Our Contributions

In this work, we solve the following two problems.

Problem 1. *Given a set R of n integer points that lie on an $n \times n$ grid, we preprocess them into a data structure such that given a query rectangle q we can efficiently report the set of maximal points in $q \cap R$.*

Problem 2. *Given a set R of n integer points that lie on an $n \times n$ grid, we preprocess them into a data structure such that given a query rectangle q we can efficiently count the number of maximal points in $q \cap R$.*

2 The Reporting Problem

In this section, we denote by $A[o, o']$, the set of elements of the array A starting from the index o to the index o'. For efficient solution of the Problem 1, we need to solve two subproblems that we discuss first. Our first subproblem is defined as

Problem 3. *We are given an array A sorted in terms of non-increasing order of the integer elements in it and is of size $O(\frac{n}{\sqrt{\log n}})$. Each element $x \in A$ belongs to one of the $\sqrt{\log n}$ smaller sub-arrays, each of size $O(\frac{n}{\log n})$. The value i for the sub-array A_i is called the tag id of the sub-array and is unique for the sub-array. For any two tag values $i, j \in [0, \sqrt{\log n} - 1]$, $A_i \cap A_j = \emptyset$. Each sub-array is sorted in terms of the non-increasing order of its elements. We need to preprocess A into a data structure such that given two indices o, o' and two tag values $a, b \in [0, \sqrt{\log n} - 1]$, we can effciently report the smallest tag t such that $a \leq t \leq b$ and $A[o, o'] \cap A_t[1, \frac{n}{\log n}] \neq \emptyset$.*

An efficient solution for the above problem will help us to emulate fractional cascading in our data structure for the Problem 1. Let $A[o] = x$ and $A[o'] = x'$. In the following section, we assume $o < o'$. Also for the tag ids a, b, we assume $a < b$. Since the array A is sorted in non-increasing order, $x' < x$. Let us first copy the elements of the sub-array $A_i \forall i = [0, \sqrt{\log n} - 1]$ into a subset S_i. Consider the interval $[x', x]$ and let $x_{mid} \in A$ be a value such that $x' \leq x_{mid} \leq x$. We split the interval $[x', x]$ into two smaller intervals $[x_{mid}, x]$ and $[x', x_{mid}]$. For the interval $[x_{mid}, x]$, suppose we have a binary string s_1 of size $\sqrt{\log n}$. The i^{th} bit of the string s_1 is set to one if the subset $S_i \cap [x_{mid}, x] \neq \emptyset$. Similarly, suppose we have a binary string s_2 for the interval $[x', x_{mid}]$. Then, to solve the Problem 3, all we need to do is the following: Take a string s_{query} of size $\sqrt{\log n}$ bits whose all the bit are set to one. As we want to find the smallest tag identity in between the range of $[a, b]$, we perform $str_{query(1)} = str_{query} << a$ (assuming $b > a$). This step sets all the bits from 0^{th} position to position $a - 1$ to zero. Next we perform $str_{query(2)} = (\sqrt{\log n} - 1 - b) >> str_{query}$ and this operation sets all the bits from position $(b+1)^{th}$ to position $\sqrt{\log n} - 1$ to zero.We set $str_{query(f)} = str_{query(1)} \wedge str_{query(2)}$. We then perform $s_{final} = (s_1 \vee s_2) \wedge str_{query(f)}$. If there are elements from the subset $S_i \forall i \in [a, b]$, the i^{th} bit of the string str_{final} will be one. Hence all we need to do is to find the least significant bit of the string str_{final} set to one. It can be seen that given we have the strings str_1 and str_2, the Problem 3 can be solved in $O(1)$ time. Next, we will see how to construct the strings s_1 and s_2. In the following section, we consider $\circ$ as concatenation operation that is $str_i \circ str_j$ means appending str_j at the end of the string str_i. Construct a binary search tree T_y whose leaf nodes are at level (height) zero and are storing the values of the array A. Each internal node stores in it the height at which it occurs in the tree T_y and the median of the values stored in the leaf nodes of the subtree rooted at the node. For each value $y_i \in A$, we will store four strings namely $str_1, str_2, str_3, str_4$ along with the value y_i in a tuple of the form $\langle y_i, str_1, str_2, str_3, str_4 \rangle$. Here, str_1, str_2 are $O(\log n)$ bits binary strings of the type of Lemma 1 each. str_3 (respectively str_4) is a binary string which is a concatenation of $O(\sqrt{\log n})$ bit strings, each of $O(\sqrt{\log n})$ bits. That is $str_3 = s_{\sqrt{\log n}} \circ s_{\sqrt{\log n}-1} \circ \ldots \circ s_1$ where $s_j \forall j = 1, \ldots, \sqrt{\log n}$ is a bit string of $O(\sqrt{\log n})$ bits. Next, consider the tree T_y whose height is at most $O(\log n)$. For any value $y_i \in A$, find the leaf node in the tree T_y storing the value y_i and then find its ancestors in the tree T_y. Let $Ance_{left}$ (respectively $Ance_{right}$) be the set of internal nodes that are left child (respectively right child) for their respective parents and are ancestors for the leaf node storing y_i. The set $Ance_{left}$ (respectively $Ance_{right}$) are sorted in terms of the increasing heights of nodes stored in the set. Let the sorted set $Ance_{left}$ be $\{\chi_1, \chi_2, \chi_3, \ldots, \chi_i\}$ (respectively $Ance_{right}$ be $\{\phi_1, \phi_2, \phi_3, \ldots, \phi_i\}$). It should be noted that $|Ance_{left}|$ (respectively $|Ance_{right}|$) is $O(\log n)$. For the value y_i we traverse the set $Ance_{left}$ (respectively $Ance_{right}$) starting from the node χ_1 (respectively ϕ_1). At each node $\chi_i \forall i = 1 \ldots |Ance_{left}|$ (respectively $\phi_i \forall i = 1 \ldots |Ance_{right}|$) we find the position for the value y_i in a sorted array B_{χ_i} (respectively B_{ϕ_i}) ordered in terms of non-increasing order of the values stored in the leaf nodes of the subtree rooted at χ_i (respectively ϕ_i). Let the position of the value y_i in B_{χ_j} (respectively in B_{ϕ_j}) be k. For the elements in the positions starting from $k+1$ to $|B_{\chi_j}|$ in the array B_{χ_j} (respectively from 1 to $k-1$ in B_{ϕ_j}), we find the distinct ids of the subsets (remember that we have copied the elements of the sub-array A_i into a subset S_i) from which these elements are coming. Let there are elements from the subsets S_1, S_2 and S_4 in positions starting from $k+1$ to $|B_{\chi_j}|$ (respectively from position 1 to $k-1$ for B_{ϕ_j}). We form a binary string s_{χ_j} (respectively s_{ϕ_j}) of size $O(\sqrt{\log n})$ bits and whose first, second and the forth bits are set to one. If $\chi_j = \chi_1$ (respectively $\phi_j = \phi_1$), set $str_3 = s_{\chi_1}$ (respectively $str_4 = s_{\phi_1}$). Else append str_3 to s_{χ_j} by performing $s_{\chi_j} \circ str_3$ (respectively str_4 to s_{ϕ_j}). This concatenated string is our new str_3 (respectively str_4). Let the level (height) of the node $\chi_j \in T_y$ ($\phi_j \in T_y$) be h. We set the h^{th} bit of str_1 to one (respectively str_2 to one). We move to the node χ_{j+1} (respectively ϕ_{j+1}). Notice that any element present in the sorted array B_{χ_j} will also present in $B_{\chi_{j+1}}$ as the node χ_{j+1} is an ancestor of χ_j. Repeat similar steps and form the string $s_{\chi_{j+1}}$ (respectively $s_{\phi_{j+1}}$). If $s_{\chi_j} \neq s_{\chi_{j+1}}$ (respectively $s_{\phi_j} \neq s_{\phi_{j+1}}$), perform $s_{\chi_{j+1}} \circ str_3$ (respectively $s_{\phi_{j+1}} \circ str_4$). Also, find the height l for the node χ_{j+1} (respectively ϕ_{j+1}) and set the l^{th} bit of str_1 to one(respectively the l^{th} bit of str_2 to one). Else, simply set $s_{\chi j+1} = s_{\chi_j}$ (respectively $s_{\phi j+1} = s_{\phi_j}$) and move to the next node χ_{j+2} (respectively ϕ_{j+2}) of the set $Ance_{left}$ ($Ance_{right}$). Repeat the process until all the nodes in $Ance_{left}$ (respectively $Ance_{right}$) are visited. Append $\sqrt{\log n}$ zeros to the end of str_3 and str_4.

Lemma 2. *For any value $y_i \in A$, the size of the string str_3 (respectively str_4) is $O(\log n)$ bits.*

Proof: A new substring s_{χ_i} (for $\chi_i \in Ance_{left}$) is included to the string str_3 only when s_{χ_i} differs from $s_{\chi_{i-1}}$ in at least one bit position. This means s_{χ_i} is included in the string str_3 when there is an element from a new subset in the array B_{χ_i} and there were no other elements from the same subset in any other array traversed previously before traversing B_{χ_i}. As the number of subsets is $O(\sqrt{\log n})$, the maximum number of substrings that can be included in str_3 is $O(\sqrt{\log n})$. Also each substring is of size $O(\sqrt{\log n})$ bits and hence the string str_3 is of $O(\log n)$ bits. □

Lemma 3. *The storage space needed for the data structure for the Problem 3 is $O(|A|)$ words.*

Proof: Once the tuples for each of the values in the array A are created, we simply maintain the array A and the tree T_y. For each node $\mu \in T_y$, all the non-increasing sorted arrays B_μ mentioned in the previous section can be deleted. Hence the claim. □

Lemma 4. *Given two indices i, j of the array A, the range $[a, b]$ $a, b \in [0, \sqrt{\log n} - 1]$ and an $O(\log n)$ bit binary string whose all the bits except the least significant $\sqrt{\log n}$ bits are set to zero, we can report the smallest tag $t \in [a, b]$ such that $S_t \cap A[i, j] \neq \emptyset$ in $O(1)$ time.*

Proof: Let $A[i] = x$, $A[j] = x'$ and $x' < x$. Find the least common ancestor $w \in T_y$ for x, x'. As the array A is sorted in non-increasing order, the value x is present in the subtree rooted at the left child of w. Let the value stored at w is x_{med}. Notice that $[x', x] = [x', x_{med}] \cup [x_{med}, x]$. Let the height of the node w be h. For the value x, we find $val_1 = rank(str_1, h-1)$ which gives us the number of substrings included in the string str_3 for x up to height $h-1$. If val_1 is 3, it means that the substring s_3 of the string $str_3 = s_{\sqrt{\log n}} \circ s_{\sqrt{\log n}-1} \circ \ldots s_3 \circ s_2 \circ s_1 \circ 000\ldots000$ has information of all the distinct sets whose elements are present in the the interval $[x_{med}, x]$. We compute $s_1 = val_1 \times \sqrt{\log n} >> str_3$ for the value x. This sets $str_3 = 000\ldots000 \circ s_{\sqrt{\log n}-1} \ldots s_{\sqrt{\log n}-2}, \ldots s_5 \ldots s_4 \ldots s_3$. Similarly compute $s_2 = (rank(str_2, h-1) \times \sqrt{\log n}) >> str_4$ for the value x'. Next, we compute $str_{query(1)} = str_{query} << a$, $str_{query(2)} = (\sqrt{\log n} - 1 - b) >> str_{query}$ and $str_{query(f)} = str_{query(1)} \wedge str_{query(2)}$. We then compute $str_{final} = (s_1 \vee s_2) \wedge str_{query(f)}$. If there is any element from the subset $S_i \forall i \in [a, b]$, the i^{th} least significant bit of str_{final} is set to one. To find the smallest tag t, we need to find the least significant bit set to one. This can be done in $O(1)$. Since all the operations (AND, OR, LEFT-SHIFT,RIGHT-SHIFT) are performed in strings of $O(\log n)$ bits, these operations can be done in $O(1)$ time. $\square$

Corollary 1. *If $str_{final} = 0$, then there are no elements from any set $S_t \forall t \in [a, b]$ in $A[i, j]$ that is $S_t \cap A[i, j] = \emptyset$.*

Our second subproblem is defined as follows.

Problem 4. *We are given an array A sorted in terms of non-increasing order of the integer elements in it and is of size $O(\frac{n}{\sqrt{\log n}})$. Each element $x \in A$ belongs to one of the $\sqrt{\log n}$ smaller sub-arrays, each of size $O(\frac{n}{\log n})$. The value i for the sub-array A_i is called the tag id of the sub-array and is unique for the sub-array. For any two tag values $i, j \in [0, \sqrt{\log n} - 1]$, $A_i \cap A_j = \emptyset$. Each sub-array is sorted in terms of the non-increasing order of its elements. We need to preprocess A into a data structure such that given an index i of the array A and a tag id t, we can efficiently find the largest value $A_t[m] \leq A[i]$.*

We use the idea of [8] to solve the problem. With each element of A, store the tag of the sub-array to which it belongs. We create $\sqrt{\log n}$ bit strings of Lemma 1 denoted by $b_0, \ldots, b_{\sqrt{\log n}-1}$. Each bit string is of size $\frac{n}{\sqrt{\log n}}$ bits. For the string b_j, we set the i^{th} bit of b_j is to 1 if the i^{th} element of the array A is from the sub-array A_j, else it is set to 0.

Lemma 5. *By storing $\sqrt{\log n}$ bit strings, each of $O(\frac{n}{\sqrt{\log n}})$ bits, we can solve Problem 4 in $O(1)$ time.*

Proof: Given an index i and the tag t, we first check if the element $A[i]$ is from the sub-array A_t. Let this be not the case. Then we consider the string b_t and perform $rank(b_t, i)$. The result gives us the number of elements from the sub-array A_t in $A[1, i]$. Hence the largest element smaller than $A[i]$ in A_t is at position $m = rank(b_t, i) + 1$. This can be done in $O(1)$ time. $\square$

We now proceed to solve the Problem 1.

2.1 Preprocessing for the Reporting Problem

Preprocessing Phase 1: Construct a tree T_x whose all the leaves are at the same level (height) and are sorted in non-decreasing order of the x coordinates of the points of the set R. Each internal node $\mu \in T_x$ has $O(\sqrt{\log n})$ children. This reduces the height of the tree to $O(\frac{\log n}{\log \log n})$. The children of a node are numbered with the left most child being $\sqrt{\log n} - 1$ and the right most child being 0. See Figure 1. Each internal node $\mu \in T_x$ is assigned an interval $int(\mu)$ which is equal to the union of the discrete intervals of the leaf nodes present in the subtree rooted at μ. Each internal node μ will have an auxiliary array A_μ which will store the y coordinates of the leaf nodes present in the subtree rooted at μ in non- increasing order. Clearly, $A_\mu = \bigcup_{i=0 \ldots \sqrt{\log n}-1} A_i$ where i is a child of μ. Each element of $A_i, i = 0, \ldots, \sqrt{\log n} - 1$ will point to its corresponding position in the array A_μ. However there will be no pointers from the elements of the array A_μ to the elements in the arrays of its children. Along with each element $y_i \in A_\mu$ store the tag number t of the child node in whose subtree y_i belongs. At the node μ, construct the data structure of Lemma 3. Also construct a Van emde Boas Tree [3] on the values of the array A_μ.

Preprocessing Phase 2:
We also keep a look up table of size $\sqrt{\log n}$ where given a value u which is a power of two that is $u = 2^i \; \forall i = 0, \ldots \sqrt{\log n} - 1$, the look up table can return the position of the maximum significant bit (i) set to one. The hash table of [4] can be used for the purpose. Also, maintain a range maxima data structure $RM_{A(\mu)}$ (see [9]) such that given two indices i, j of A_μ, we can return the maximum x coordinate among the points whose y coordinates are stored between $A_\mu[i]$ to $A_\mu[j]$.

Preprocessing Phase 3:
The following step is to be executed in all internal nodes. Let $\mu \in T_x$ is an internal node. Create a data structure of Lemma 5 at the node μ for the values of A_μ as $A_\mu = \bigcup_{i=0,\ldots,\sqrt{\log n}-1} A_i$. Here i is a child of μ. For the auxiliary array at the root node A_{root}, there is an exception. Each element of the array A_{root} has $2\sqrt{\log n}$ pointers, the first $\sqrt{\log n}$ pointers (respectively the other $\sqrt{\log n}$ pointers) are pointing to the largest elements (respectively smallest elements) smaller (respectively greater) than the concerned element in each of the arrays associated with the children of the root node.

Lemma 6. *The storage space needed by the above discussed data structure is* $O(n \frac{\log n}{\log \log n})$ *words.*

2.2 Query Algorithm

Given a query rectangle $q = [x_1, x_2] \times [y_1, y_2]$, we allocate the segment $[x_1, x_2]$ to the internal node $\mu \in T_x$ if $int(\mu) \subseteq [x_1, x_2]$ but $int(parent(\mu)) \not\subseteq [x_1, x_2]$. Let the nodes colored black in Figure 1 are the canonical nodes to whom the segment $[x_1, x_2]$ is allocated. The query q will be accompanied by a $O(\log n)$ bit string denoted by str_{query} whose $\sqrt{\log n}$ least significant bits are set to one and all the other bits are set to zero. As specified in [6], an interval $[x_1, x_2]$ can be represented as a union of intervals assigned to some nodes $v_1, \ldots, v_k$ that can be grouped into $\frac{\log n}{\log \log n}$ groups $G_1, \ldots, G_h$. Each group G_i contains a set of children $v_{l_i}, v_{l_{i+1}}, \ldots, v_{l_r}$ for some node v_l. There are at most two groups in each level. Hence there are $O(\frac{\log n}{\log \log n})$ groups.

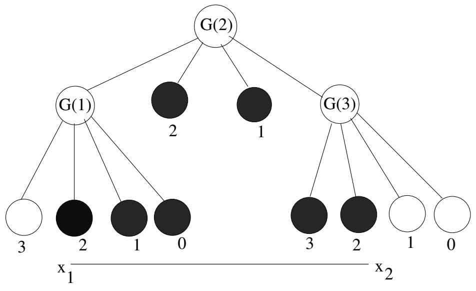

Fig. 1. The black nodes are the canonical nodes to whom the interval $[x_1, x_2]$ for the rectangle $q = [x_1, x_2] \times [y_1, y_2]$ is allocated

The nodes marked $G(1), G(2), G(3)$ in Figure 1 are such groups. With reference to Figure 1, we should begin searching for maximal points starting from node marked 2 which is a child of the node marked $G(3)$ as any point allocated to the node 2 of the node marked $G(3)$ will have its x coordinate greater than any other point in the rest of the canonical nodes. However, we are not sure if there is any point inside the rectangle q from the node marked 2. Hence, we do the following: We first find the two indices $[i, j]$ in the array $A_{G(2)}$ such that $A_{G(2)}[i]$ is the largest value smaller than or equal to y_2 and $A_{G(2)}[j]$ is the smallest value greater than or equal to y_1. This can be done using the Van emde Boas tree. Next, by using the technique of Lemma 5, we find the indices i, j in the auxiliary array $A_{G(3)}$ of the node $G(3)$. At the node $G(3)$, by using the technique of Lemma 4 we find the smallest tag $t \in [2, 3]$ such that there is at least one element from the array $A(t)$ in $A_{G(3)}[i, j]$. Here t is the tag for a child of the node marked $G(3)$. Let $t = 2$ and hence by using the technique of Lemma 5, we fix the two indices i', j' of the array $A(2)$ at the node marked 2 which is a child of the node marked $G(3)$. By using the range maxima data structure $RM_{A(2)}$ constructed in preprocessing phase 2, we find the index l $(i' \leq l \leq j')$ of the point having the largest x coordinate among all the points whose y coordinates are stored in $A_2[i', j']$. Next, we perform range maxima query among the elements in $A_2[i', l]$. We keep repeating the step as long as there are maximal points from the array A_2 in the rectangle q. Let the last maximal point from the node 2 be $p = (p_x, p_y)$. Find the position of p_y in the array $A_{G(3)}$. This can be done in $O(1)$ time as there are pointers from the elements in the array of a child node to its respective position in the parent array. Notice that by Corollary 1 if there are no points from the arrays A_2 and A_3 in $A_{G(3)}[i, j]$, the string str_{final} will be zero and in that case, we will move to the node marked $G(2)$. Keep repeating similar steps until we have visited all the group nodes.

Lemma 7. *The maximal point sets in the query rectangle can be reported in* $O(\frac{\log n}{\log \log n} + k)$ *time where* k *is the size of the output.*

Proof: It is evident from the query algorithm and the Corollary 1. □

Theorem 1. *Given a set* R *of* n *integer points from the grid* $[1, n] \times [1, n]$*, we can construct a data structure of* $O(n\frac{\log n}{\log \log n})$ *words such that given a query rectangle* q*, we can report the maximal points in* $q \cap R$ *in* $O(\frac{\log n}{\log \log n} + k)$ *time where* k *is the number of maximal points in* $q \cap R$*.*

3 The Counting Problem

Consider the data structure for the Problem 1. At each internal node $\mu \in T_x$, we have an auxiliary array A_μ which stores the y coordinates of the points present in the leaf node of the subtree rooted at μ. The points are stored in non-increasing order of y coordinates in the array A_μ.

3.1 Preprocessing for the Counting Problem

1. Construct a balanced binary search tree T_y on the values of A_μ.
2. Consider a node $\phi \in T_y$ and associate an auxiliary array A'_ϕ which stores the y coordinates of the leaf nodes of subtree rooted at ϕ in their non-increasing order.
3. If ϕ is a left node of the tree T_y, we do the following:
   (a) Copy the first element of the array A'_π and store it as the last element of the array A'_ϕ. Here π is the right sibling of ϕ in T_y.
   (b) Let the y coordinate of the last element of the array A'_ϕ (that is the element copied from A'_π) be p'_y. For each point $p = (p_x, p_y) \in A'_\phi$, find the point p'' of the array A'_ϕ which has the maximum x coordinate in the 3-sided rectangle of the form $[p_x, \infty) \times [p'_y, p_y]$.
   (c) Next, form an axes parallel rectangle of with p, p'' as the diagonal points and find the number of maximal points (including p and p'') inside the rectangle. We store that value in a variable max_p
   (d) Also, for each point $p \in A'_\phi$, we find the point $p_{crux} \in A'_\pi$ such that p_{crux} lies in the south east quadrant of p and the y coordinate of p_{crux} is just below the y coordinate of p. Here π is the right sibling of ϕ.
   (e) We create a tuple $\langle p, p'', m, max_p\rangle$ and store it along with the point p in the array A'_ϕ. Here m is the index of the point p_{crux} for the point p'' in the array A'_π. Also π is the right sibling of ϕ.
   (f) However, if $p'' = A'_\pi[0]$, we set $max_p = max_p - 1$ and then we store the tuple $\langle p, A'_\pi[0], 0, max_p\rangle$. Remember the first element of array A'_π has been copied as the last element of A'_ϕ.

   (g) If there are no points in $[p_x, \infty) \times [p'_y, p_y]$,
      i. Check if the last point of A'_ϕ is in the south east quadrant of the point p. If "YES", set $max_p = 1$. Store the tuple $\langle p, A'_\pi[0], 0, max_p \rangle$ as tuple.
      ii. Else set $max_p = 0$, and store the tuple $\langle p, p, m, max_p \rangle$ where $A'_\pi[m]$ has the point which is just below the point p in the south east quadrant of p. In other words, m is the index of the point p_{crux} for the point p in the array A'_π.

4. Now, consider the node π which is a right node in the tree T_y. For each point $p \in A'_\pi$, check if the first point p' of the array A'_π (the one which is copied to the left sibling) is in the north west quadrant of p.
   (a) If "YES", store the number of maximal points that are there in the axes parallel rectangle formed by taking p, p' as diagonally opposite elements in a variable max_p and store it along with p in the array A'_π.
   (b) If "NO", do nothing.

Lemma 8. *The storage space needed by T_y is $O(n \log n)$ words. The total storage space needed by the complete data structure is $O(n \frac{\log^2 n}{\log \log n})$ words.*

3.2 Query Algorithm

Let the query rectangle be $[x_1, x_2] \times [y_1, y_2]$. By using the ideas of the Problem 1, we ensure that we visit only those canonical nodes which which contribute to the set of maximal points inside the query rectangle. Let $\mu \in T_x$ be such a node (wrt to Figure 1, μ is one of the nodes colored black). Consider the array A_μ. Any point allocated to the node μ has its x coordinate overlapping with $[x_1, x_2]$. Let the points in $A_\mu[i, j]$ be the points whose y coordinates are in between $[y_1, y_2]$. We will have to calculate the number of points between indices i, j which are in the maximal point set. Let the k^{th} index of the array A_μ $(i \leq k \leq j)$ stores the point with the largest x coordinate among all the points from the node μ inside the query q. Call this point as p and the point in the i^{th} index as p'. Notice that the point p and p' are sure to be in the maximal point set. We need to calculate the number of maximal points between p and p' in the query rectangle which we do as follows:

1. Find the least common ancestor (LCA) for the point p and p' in the tree T_y.
2. Let w be the LCA and let u and v be the left and the right child of w respectively.
3. Find the position of the index i in A'_u and the position of the index k in A'_v.
4. Consider the tuple $\langle p', p'', m, max_{p'} \rangle$ stored along with the point p' in the array A'_u.
5. If $p'' = A'_v[0]$, then the number of maximal point set is equal to $max_{p'} + max_p$. Return the value and Exit.
6. Else, move to the index m of A'_v .
7. If $max_{p'} \neq 0$, then

   (a) If $A'_v[m] \neq p$ then the number of maximal points is $(max_{p'}) + (max_p - max_{p_{crux}})$.
   (b) If $A'_v[m] = p$, then then the number of maximal points is $(max_{p'}) + 1$.
8. If $max_{p'} = 0$, then
   (a) If $A'_v[m] \neq p$ then the number of maximal points is $(max_p - max_{p_{crux}}) + 1$.
   (b) If $A'_v[m] = p$, then then the number of maximal points is two.

In consistence with the query algorithm, let p and p' be respectively the points with maximum x and y coordinates among all the points from the node μ inside the query rectangle q. Let w be the split node (LCA) for the points p, p' in the tree T_y. The point p is in A'_v and p' is in A'_u where v and u are respectively the right and the left children of w. Let us call the first point of the array A'_v as p'''.

Lemma 9. *If p and p' are inside q and w is the split node (LCA), then p''' is inside q.*

The above lemma justifies the step 4(b) of the preprocessing that is why we do nothing if p''' is not in the north west quadrant of p in the array A'_v. After the above lemma, we have the following two cases namely: (i) $p'_x \leq p'''_x \leq p_x$ and (ii) $p'''_x \leq p'_x \leq p_x$. We will start with Case (i). Case (ii) can be handled similarly.

3.3 Case (i): $p'_x \leq p'''_x \leq p_x$

Notice that by step 3(a), the point p''' is copied as the last element of the array A'_u. By step 3 (b) of the preprocessing, we find out the maximal point with the maximum x coordinate among all the points in A'_u in the 3-sided rectangle $[p'_x, \infty) \times [p'_y, p'''_y]$. This again leads to three possible cases namely:
Case i(a) there are points inside the three sided rectangle and the point p''' does not have the maximum x coordinate inside the three sided rectangle
Case i(b) there are no points inside the three sided rectangle and
Case i(c) there are points inside the three sided rectangle and the point p''' has the maximum x coordinate inside the three sided rectangle.
We will discuss Case i(a) elaborately. Case i(b) and Case i(c) can be managed in a similar way.

Case i(a): Suppose, the first situation is true that is there are points inside the three sided rectangle and the point p''' does not have the maximum x coordinate inside the three sided rectangle. Then there is a point p'' with the maximum x coordinate in the 3-sided rectangle. We discuss some properties of p'' in Lemma 10.

Lemma 10. *The point p'' will be in array A'_u and p'' will be a maximal point inside the query rectangle.*

As per the step 3(d) of the preprocessing algorithm, for the point p'', we will have a point $p_{crux} \in A'_v$ such that p_{crux} is in the south east quadrant of p'' and the y coordinate of p_{crux} is just below p''. We discuss some properties of p_{crux} in Lemma 11 and 12.

Lemma 11. *The point p_{crux} will always exist for the point p''. The x coordinate of p_{crux} has to be less than or equal to the x coordinate of the point p that is $p_{crux,x} \leq p_x$.*

Proof: Let us assume that $p_{crux,x} > p_x$ is true. Since $p''_y \in [c,d]$ and $p_y \in [c,d]$, $p_{crux,y} \in [c,d]$. Also $p_{crux_x} \in [a,b]$, by default. Hence $p_{crux} \in q$. But, then the point p cannot be the point with the maximum x coordinate inside q. A contradiction. □

Lemma 12. *Consider an axes parallel rectangle with p and p''' being diagonally opposite corners. Let p_{crux} be inside the rectangle. Then p_{crux} will be a maximal point inside the rectangle.*

Proof: If there is any point dominating p_{crux}, then p_{crux} simply cannot be the point whose y coordinate is just below p''_y in the $SE(p'')$. □

Lemma 13. *If there are points in $[p'_x, \infty) \times [p'_y, p'''_y]$ and p''' does not have the maximum x coordinate, then our algorithm correctly counts the number of the maximal points inside the query rectangle.*

Proof: Consider the axes parallel rectangle with p, p' being diagonally opposite elements. Denote it as $Rect_{p,p'}$. Let the number of maximal points inside $Rect_{p,p'}$ be denoted as $|Rect_{p,p'}|$. Notice that $|Rect_{p,p'}| = |Rect_{p,p_{crux}}| + |Rect_{p',p''}|$. Notice that the step 7 (a) of the query algorithm is doing exactly the same thing. On the other hand if $p_{crux} = p$ for the point p'', then there are no points inside $Rect_{p'',p}$ and hence the number of maximal points in $Rect_{p,p'}$ is equal to $|Rect_{p',p''}|$ plus one (one for the point p) and this what step 7 (b) is doing. □

Case i(b)

Lemma 14. *Suppose there are no points inside the 3-sided rectangle $[p'_x, \infty) \times [p'_y, p'''_y]$, but p''' is in the south east quadrant of p'. Then, our algorithm can correctly compute the number of maximal points inside the query rectangle.*

Proof: In this case, p''' has the maximum x coordinate inside the 3-sided rectangle and step 5 of our algorithm handles the situation. □

Case i(c):

Lemma 15. *Suppose there are points inside the 3-sided rectangle $[p'_x, \infty) \times [p'_y, p'''_y]$, but p''' is the point with the maximum x coordinate inside the three sided rectangle. Then, our algorithm can correctly compute the number of maximal points inside the query rectangle.*

Proof : Notice that when there are points inside the three sided rectangle but p''' has the maximum x coordinate, then for the point p' we do have $p'' = p'''$ and hence by step 3 (f) of preprocessing, we will have the tuple $\langle p', p''', 0, max_{p'} - 1\rangle$. Here $max_{p'}$ is equal to the number of maximal points inside the axes parallel rectangle with p', p''' as diagonally opposite elements that is $Rect_{p',p'''}$. Notice that $A'_v[0] = p'''$, where v is the right sibling of u. During the execution of the query algorithm, step (5) will deal with the situation and the number of maximal points that will be returned is $max_{p'} - 1 + |Rect_{p''',p}|$. □

3.4 Case (ii): $p'''_x \leq p'_x \leq p_x$

With arguments similar to Case (i), it can be shown that the Case (ii) can be handled by our algorithm and hence we skip the discussion.

Theorem 2. *Given a set R of n integer points from the grid $[1, n] \times [1, n]$, we can construct a data structure of size $O(n\frac{\log^2 n}{\log \log n})$ words such that given a query rectangle q, we can report the number of maximal points in $P \cap q$ in $O(\frac{\log^{\frac{3}{2}} n}{\log \log n})$ time.*

Proof: The maximum number of canonical node from which the maximal points is $O(\frac{\log^{\frac{3}{2}} n}{\log \log n})$. At each node we take $O(1)$ unit of time. Hence the total running time for counting the maximal points inside the query rectangle is $O(\frac{\log^{\frac{3}{2}} n}{\log \log n})$. □

References

1. Brodal, G.S., Tsakalidis, K.: Dynamic Planar Range Maxima Queries. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 256–267. Springer, Heidelberg (2011)
2. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage (extended abstract). In: SODA, pp. 383–391 (1996)
3. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. 6(3), 80–82 (1977)
4. Fredman, M.L., Komls, J., Szemerdi, E.: Storing a sparse table with o(1) worst case access time. In: FOCS, pp. 165–169 (1982)
5. Kalavagattu, A.K., Das, A.S., Kothapalli, K., Srinathan., K.: On finding skyline points for range queries in plane. In: CCCG, pp. 343–346 (2011)
6. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Comput. Geom. 42(4), 342–351 (2009)
7. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
8. Yu, C.C., Hon, W.K., Wang, B.F.: Improved data structures for the orthogonal range successor problem. Comput. Geom. 44(3), 148–159 (2011)
9. Yuan, H., Atallah, M.J.: Data structures for range minimum queries in multidimensional arrays. In: SODA, pp. 150–160 (2010)