General Purpose Index-Based Method for Efficient MaxRS Query

Xiaoling Zhou[1(✉)], Wei Wang[1], and Jianliang Xu[2]

[1] University of New South Wales, Sydney, Australia
{xiaolingz, weiw}@cse.unsw.edu.au
[2] Hong Kong Baptist University, Hong Kong, China
xujl@comp.hkbu.edu.hk

Abstract. The Maximizing Range Sum problem is widely applied in facility locating, spatial data mining, and clustering problems. The current most efficient method solves it in time $O(n \log n)$ for a particular given rectangle size. This is inefficient in cases where the queries are frequently called with different parameters. Thus, in this paper, we propose an index-based method that solves the maxRS query in time $O(\log n)$ for any given query. Besides, our method can be used to solve the k-enclosing problem in time $O(1)$ for any given k value if indexes are sorted according to the optimizing criteria, or $O((n-k)^2k + n \log n)$ without using any index, which is comparative to the current most efficient work.

Keywords: Maximizing range sum · Index construction · Query processing

1 Introduction

In this paper, we study the *Maximizing Range Sum* (maxRS) problem [1–5], which is also known as the *Maximum-enclosing Rectangle Problem* in computational geometry. Given a set of n points P, each with a positive weight $w(p)$, the maxRS problem finds the placement of an axis-parallel rectangle r of given size $\alpha \times \beta$ that maximizes the weight sum of points covered by r. The maxRS problem is well-motivated in recent work [3–5], and is generally applied in facility location problems [14] for finding the best facility location with maximum number of potential clients, spatial data mining for extracting interesting locations from log data [19], and point enclosing problems.

However, existing work targeted at finding efficient algorithms to answer the query given a particular rectangle size, and they need query time superlinear in n. Currently, the best method solves the exact maxRS problem in time $O(n \log n)$ [2] based on the plane-sweep algorithm [1], and the best algorithm for the $1-\epsilon$ approximate version of the problem works in time $O(n \log \frac{1}{\epsilon} + n \log \log n)$ via grid sampling. This is undesirable in cases where the maxRS queries are asked frequently with different parameters. For example, in computational geometry, maxRS serves as a subroutine and is called many times with different rectangle

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part I, LNCS 9827, pp. 20–36, 2016.
DOI: 10.1007/978-3-319-44403-1_2

Table 1. Comparison of algorithms in terms of space-time tradeoffs

Algorithm	Query	Space	Comment
[2]	$O(n \log n)$	$\Theta(n)$	Exact
[4]	$O(n \log \log n)$	$O(n)$	$(1-\epsilon)$ approximate with $\geq 1 - \frac{1}{n}$ probability
This paper	$O(\log n)$	$\Theta(n \cdot \lambda) = O(n^3)$	Exact
This paper	$O(\log n)$	$\Theta(\log n \cdot \lambda) = O(n^2 \log n)$	$(1-\epsilon)$ approximate guaranteed

Notes: (1) λ is the maximum size of any k-line, and (2) assume ϵ is a constant.

sizes in point enclosing problems [16]; and in real life applications, a web service that answers queries like finding the location in a city with most number of tourist attractions within a given reachable area, will be enquired by tons of users with different parameters.

Therefore, in this paper, we propose new solutions to this problem that answer queries much more efficient than previous methods, by making use of a special precomputed index. The idea of using precomputed index to accelerate query processing is a common one in databases, and our study expands the spectrum of space-time tradeoff to the maxRS problem. In addition, our method immediately gives comparable or superior results to other related problems compared with existing solutions.

The main idea of our method is to try to precompute and store as few as possible the maxRS results for a limited number of rectangle sizes, and yet still be able to answer any arbitrary query in an efficient manner. Based on the index, we answer the maxRS query in $O(\log n)$ time, which compares favorably with existing bond $O(n \log n)$. Besides, our method solves the k-enclosing rectangle problem in constant time if the index is sorted according to the optimizing criteria, or $O((n-k)^2 k + n \log n)$ without using index, which is comparative to the best time bound achieved so far [11] for general k values. We solve the k-enclosing problem for all possible k values in time $O(n^3 \log n)$, while the direct adaption of existing method [11] takes time $O(n^4)$. On top of the above, our index can be used to answer maximum point enclosing problems with $O(\log n)$ query time. Details are presented in Sect. 6.

We highlight our main contributions in the following:

- We design an index structure that supports very fast maxRS queries. The index is based on novel concepts of changing points and k-lines. The method provides the possibility of efficient batch query, and space-time tradeoff for the maxRS problem (Sect. 3).
- We design a query processing technique achieves time $O(\log n)$ by nontrivially adapting the idea of Fractional Cascading based on a tree structure (Sect. 4).
- We present applications of our method on other problems, and the superiority of our method compared with existing works (Sect. 6).
- We perform experiments on synthetic as well as real datasets to show the feasibility and efficiency of our methods compared with the state-of-the-art methods (Sect. 7).

2 Related Work

The MaxRS Problem. Nandy et al. [2] proposed an $O(n \log n)$ time algorithm to solve the maxRS problem using the plane-sweeping technique [1] with interval trees. Choi et al. [3] proposed an external memory solution following the distribution sweep paradigm [7], and the work was further extended in [5] by providing solutions to the AllMaxRS problem, which retrieves all the locations of rectangles achieving the maximum total covered weight. Recognized the need for further speedup, Tao et al. [4] studied the approximate MaxRS problem, and obtained a $(1 - \epsilon)$-approximate answer with high confidence in time $O(n \log \frac{1}{\epsilon} + n \log \log n)$ via grid sampling. Instead of finding a rectangle with maximum cover weight, Das et al. [6] retrieved the highest density axis-parallel rectangle r where density was defined as the ratio between the covered weight of r and its area.

All the above works solved the exact or approximate maxRS problem for a fixed query rectangle size only, required at least query time superlinear in n, and they did not consider using an index. Our work is the first that builds a special index to speed up the query processing time to $O(\log n)$, hence provides a different solution exploiting the space-time tradeoff and is beneficial to frequent queries.

k-Enclosing Problem. Driven by the wide application in pattern recognition [13], facility locating [14], and VLSI chip design [18], the problem of finding the axis-aligned rectangle, square, or circle that encloses k of n points in $\mathcal{R}^2$ and is optimal on some particular criteria (area, perimeter, or diameter etc.) has been long studied in a large number of works [8–12,15,16]. Aggarwal et al. [8] obtained the smallest k-enclosing rectangle or square in time $O(nk^2 \log n)$ and space $O(nk)$ based on higher-order Voronoi diagrams in 1991. The complexity was improved several times by subsequent works [9–11]. The work in [10] improved the time complexity to $O(nk^2 + n \log n)$ using space $O(n)$ based on the idea of testing sets of the $O(k)$ nearest neighbors to each point. Segal et al. [11] solved the problem in time $O(n + k(n - k)^2)$ using $O(n)$ space, which performed better than previous methods on large k values, and they also proposed solution on d (> 2) dimensions that takes time $O(dn + dk(n-k)^{2(d-1)})$ and space $O(dn)$.

Datta et al. [9] solved the smallest k-enclosing square problem in $O(n \log n + n \log^2 k)$ time and $O(n)$ space. The bound was then improved to $O(n + (n - k) \log^2(n - k))$ using $O(n)$ space by Mahapatra et al. [16], which searched the target square by means of prune and search technique and adopted the maxRS problem as a subroutine to guide the search. Das et al. [17] addressed the generalized version of above problem, where the desired rectangle may be of arbitrary orientation, and their method runs in $O(n^2 \log n + kn(n-k)(n-k+\log k))$ time using $O(n)$ space.

3 The New Index

We formally define the problem.

Definition 1. *Let P be a set of n points*[1] *in 2D Euclidean space $\mathcal{R}^2$, and each point $p \in P$ carries a positive value $w(p)$ as its weight. Given non-negative values α and β, the goal of the maximizing range sum (maxRS) problem is to place a $\alpha \times \beta$ rectangle r in $\mathcal{R}^2$ to maximize the covered weight of r, defined as: covered-weight$(r) = \sum_{p \in P \cap r} w(p)$.*

Note that the rectangles can be placed at any position. We denote $maxRS(\alpha, \beta)$ as the maximum total weight that can be achieved given query (α, β).

In this paper, we deal with points with equal weights, a.k.a., the *max-enclosing problem*. Our method can be easily extended to handle general weighted cases (See Sect. 6).

Notations. We introduce some notations used in the rest of the paper. Let $P = \{ p_1, p_2, \ldots, p_n \}$ denotes the given n data points. $p_i.x$ (resp. $p_i.y$) denotes the x-coordinate (resp. y-coordinate) of point p_i. We can sort P into P^x by the x-coordinates of the points, and obtain P^y analogously. For a pair of different points p_i and p_j $(i \neq j)$, we can obtain an x-interval $a = |p_i.x - p_j.x|$. We have $O(n^2)$ distinct x-intervals, and they are collectively denoted as list A. We obtain list B of all distinct y-intervals analogously. Given an x-interval a_i, and the two points p_l and p_r $(l, r \in [1, n])$ in P that form this x-interval (w.l.o.g., assume $p_l.x < p_r.x$), we denote the list of points $p_j \in P$ such that $p_l.x \leq p_j.x \leq p_r.x$ sorted according to y-coordinate as S_{a_i}. Note that p_l and p_r are included in S_{a_i}. More notations will be introduced at their first use in the rest of the paper.

3.1 A Naïve Solution

A naïve idea is to index the maxRS values for *all possible* queries. While there are infinite number of possible queries, Lemma 1 divides them into $O(n^4)$ *equivalent classes*, and this immediately leads to a naïve index and query processing method.

Lemma 1. *Given a query (α, β), let a^* be the largest value in A such that $a^* \leq \alpha$, and define b^* analogously. Then $maxRS(\alpha, \beta) = maxRS(a^*, b^*)$.*

The index is essentially an $O(n^2) \times O(n^2)$ matrix, which stores the precomputed maxRS values for every $(a_i, b_j) \in A \times B$. An example dataset and its naïve index matrix is shown in Fig. 1(a), (b). We call the matrix cell (a^*, b^*) as the *target cell* of the query. The values of a^* and b^* can be obtained using binary search in A and B, respectively. Therefore, the index size is $O(n^4)$, and the query time is $O(\log n)$.

3.2 Index the Changing Points into *k*-Lines

The naïve index contains many redundant values. Our important observation is that maxRS values in the matrix follow certain pattern and can be exploited to further reduce the index size.

[1] W.l.o.g., we assume that no two points have the same x (or y) coordinate.

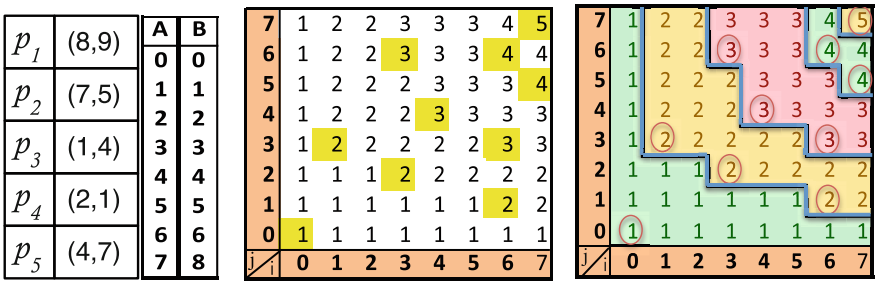
Definition 2 (Changing Point). *A changing point (CP) in the matrix is a cell $M[i,j]$ such that (1) $M[i-1,j] = M[i,j-1] = M[i-1,j-1]$ if any of them exists, and (2) $M[i,j] > M[i-1,j]$.*

We highlight all the CPs in the example dataset in Fig. 1(b).

We observe that all the CPs with same maxRS value k collectively form a *skyline*, which we call **k-line**. There are n number of k-lines, and they divide the maxRS matrix into n separate regions, as demonstrated as shaded regions of different colors in Fig. 1(c). Our new index just stores these n k-lines (each is organized as a list of CPs sorted on the x-axis). This gives us an index of size $O(n \cdot \lambda)$, where λ is the maximum size of any k-line.

Lemma 2. *λ is at most $O(n^2)$.*

The proof is based on the observation that each CP in L_k has distinct x(or y) intervals, and the size of A and B are bounded by $O(n^2)$. Thus, the index size is upper bounded by $O(n^3)$. Note that in real practice, λ is demonstrated to be consistently linear in n in our empirical evaluation, which leads to cn^2 index space, where c is a constant. Hence, the index size is significantly smaller than the naïve index.



(a) Dataset (b) MaxRS Matrix (c) Changing Points

Fig. 1. Index (Color figure online)

This new index poses challenges on query processing, as the cell of (a^*, b^*) for the query may not be a CP, hence is not stored in the index; this renders the previous $O(\log n)$ query algorithm inapplicable. Nevertheless, we devise a novel $O(\log n)$ query processing method and introduce it in Sect. 4. The detailed construction of index is presented in Sect. 5.

4 Query Processing

Next, we introduce our query processing algorithm for the new index based on k-lines. We start with an algorithm with $O(\log^2 n)$ complexity and then further improve it to $O(\log n)$ by adapting the Fractional Cascading (FC) technique.

4.1 Dominance Relationship

We first introduce a few notations and useful Lemmas.

Given a query Q with search parameters (α, β), we can map it into a two dimensional point (α, β) (called *query point*). Note that every changing point is also a valid query point, and the same mapping applies. A k-line can be mapped into a polyline with axis-parallel segments by connecting two adjacent changing points, q_1 and q_2 ($q_1.x < q_2.x$), via inserting another point $(q_2.x, q_1.y)$ in between (refer to blue lines in Fig. 1(c)). In the rest of this section, we will abuse the notation of k-line and L_k to denote the polyline it maps to. Given a k-line (L_k), it divides the first quadrant into two disjoint regions, the one on the lower-left side of it (called $LOW(L_k)$) and the other on the upper-right side (called $HIGH(L_k)$). Points on L_k is included in $HIGH(L_k)$.

We can define a **dominance** relationship between any two query/matrix points as follows.

Definition 3 ((Point) Dominance). *Let q_1 and q_2 be two points. q_1 dominates q_2, denoted as $q_1 \prec q_2$, if and only if $q_1.x \leq q_2.x \wedge q_1.y \leq q_2.y$ and $q_1 \neq q_2$.*

Lemma 3. *If $q_1 \prec q_2$, then $maxRS(q_1) \leq maxRS(q_2)$.*

We can generalize the dominance to be between a k-line and a point Q.

Definition 4. ((Line-Point) Dominance). *Given a k-line and a point q, q has to be either in $LOW(L_k)$, or in $HIGH(L_k)$. We say q dominates L_k, denoted as $q \prec L_k$ for the former case, and $L_k \prec q$ for the latter case.*

Unlike the point dominance where it is possible that two points do not dominate each other, a k-line and a point always fall into one of the two dominance orders. We also have the following Lemma.

Lemma 4. *A query point $Q \in HIGH(L_k)$ if and only if there is a changing point q in L_k such that q dominates Q.*

4.2 The $O(\log^2 n)$ Query Processing Method

Given a query (α, β), we only need to find its target cell with parameters (a^*, b^*). Note that after mapping queries and matrix cells to points, the region defined by the axes and two adjacent k-lines has the same maxRS values. Therefore, we only need to determine which region the target cell falls into. This can be performed by a standard *outer* binary search on the n disjoint regions formed by the k-lines. Technically, this requires us to determine the largest k-line that dominates the target point (a^*, b^*). This step can be performed by another *inner* binary search among the CPs of a k-line, thanks to Lemmas 4 and 5. The inner binary search takes time $O(\log \lambda) = O(\log n)$, and hence the total query cost is $O(\log^2 n)$.

In the interest of space, in the following, we focus on the inner binary search step to determine the line-point dominance.

Algorithm 1. DominanceCheck(L_i, Q, *low*, *high*)

Input : L_i is a i-th sky-line; Q is the query point.
Output: 0 if Q is found in L_i; 1 if L_i dominates Q; -1 if L_i does not dominate Q.

```
1  while high ≥ low do
2      mid ← ⌊(low + high)/2⌋; q ← L_i[mid] ;
3      if q = Q then return 0 ;
4      else if q ≺ Q then return 1 ;
5      else if Q ≺ q then return -1 ;
6      else
7          if q.x ≥ Q.x and q.y ≤ Q.y then
8              high ← mid − 1 ;
9          else                          /* must be q.x ≤ Q.x and q.y ≥ Q.y */
10             low ← mid + 1 ;
11 return -1                        /* No dominating point found ⇒ Q ≺ L_i */
```

The pseudocode of sub-routine to determine the dominance relationship between the current L_i and the query point Q is given in Algorithm 1. Initially, *low* and *high*, the index values into L_i, are set to 1 and $|L_i|$, respectively. At each binary search iteration, we take the middle point q in the current search scope, and check the dominance relationship between q and Q. There are four cases:

1. $q = Q$ (Line 3): the query point is a CP in L_i, hence we return 0. The outer binary search will terminate immediately.
2. q dominates Q (Line 4), hence L_i dominates Q, and we return 1. The outer binary search will continue to search among L_js, where $j \in [i+1, n]$.
3. Q dominates q (Line 5), hence Q dominates L_i, and we return -1. The outer binary search will continue to search among L_js, where $j \in [1, i-1]$.
4. q and Q do not dominate each other (Lines 7–10). We need to investigate this further by considering another changing point in L_i. This is handled by moving q towards Q for continuing the binary search. Intuitively, this abandons the half part of L_i that no point inside there could dominate/is dominated by Q.

Finally, if we exit the **while** loop, it means no changing points in L_i can dominate Q and vice versa, we know that Q dominates L_i in this case due to Lemma 4.

The correctness of the algorithm is provided by the following Lemma.

Lemma 5. *Algorithm 1 correctly determines the line-point dominance relationship between L_i and Q.*

An Example of Algorithm 1. Using the same example as Fig. 1, we assume the algorithm input is $Q(2,5)$ and L_3. We have $L_3 = \{(3,6),(4,4),(6,3)\}$. The search starts with the middle entry $q(4,4)$ in L_3. We cannot decide the dominance relationship between Q and q since $q.x > Q.x \wedge q.y < Q.y$, but we know the right

half of L_3 must not contain point dominates or is dominated by Q, as all the points q' on right of q have $q'.x > q.x > Q.x$ and $q'.y < q.y < Q.y$. Therefore, we search the left half of L_3 and find Q dominates entry $(3, 6)$, so -1 is returned.

4.3 The $O(\log n)$ Query Processing Algorithm

In this section, we further improve the previous query processing algorithm to achieve $O(\log n)$ complexity by adapting our problem to use the Fractional Cascading (FC) technique. FC can reduce binary searches on multiple sorted lists into one by judiciously sampling elements from other lists to one single list. A typical case is to find the smallest number in all the lists no smaller than a query number.

FC cannot be directly applied due to two major technical challenges. One is that our inner search is based on dominance checking, which is essentially 2D, while traditional FC essentially works in 1D lists as the lists need to be sorted. We overcome this by reducing our problem to 1D thanks to the following Lemma.

Lemma 6. *Given Q and L_k, $Q \in HIGH(L_k)$ (i.e. $maxRS(Q) \geq k$) if and only if q dominates Q, where q is the CP in L_k with the largest x-coordinate yet $q.x \leq Q.x$.*

We call such q the **anchor point** of Q in L_k. Then, for each L_k, the goal is to find the anchor point of Q from L_k, so that we can resolve the dominance relationship between Q and L_k.

The second challenge is that by applying FC directly, the total query cost will be $O(\log n + n)$, even worse than $O(\log^2 n)$. We overcome this by observing that we only need to check $\log n$ (rather than n) k-lines (i.e., lists). In addition, though the set of $\log n$ lists inspected will be different for different Q, the visiting order between any two lists is fixed due to the binary search procedure (e.g., the middle list is always inspected before one of the two quadrant lists). Therefore, we can construct a binary tree that reflects the binary search process on the k lists, and apply FC technique bottom up for every parent-child list pairs. We prove later that the total space is still linear in the total size of the lists.

We show the data structure created for the example dataset (Fig. 1) in Fig. 2 and explain it below:

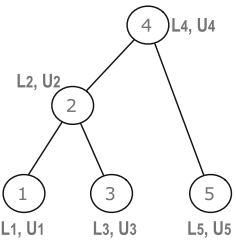
1. Construct a binary tree following the order of binary search on n values. Each node with key value i (inside the node) is logically associated with list L_i.
2. Each node actually stores a list U_i, constructed as a result of applying FC recursively in the subtree rooted at it. For a leaf node v, $U_i = L_i$. For an internal node representing L_i, $U_i = L_i \cup \mathsf{even}(U_{\text{left}}) \cup \mathsf{even}(U_{\text{right}})$, where U_{left} and U_{right} are the U list for the left and right child node of v, respectively.

Additionally, we store three pointers $[t_1, t_2, t_3]$ for each entry e in U_i (See the blue values shown in Fig. 2(a) above each element e), where t_1, t_2, t_3 point to the position of the *anchor point* of e in L_i, U_{left}, and U_{right}, respectively.

L_1	(0,0)
L_2	(1,3)(3,2)(6,1)
L_3	(3,6)(4,4)(6,3)
L_4	(6,6)(7,5)
L_5	(7,8)

U_1	(0,0)
U_2	[1,1,∅] [2,1,1] [2,1,2] [3,1,3] (1,3) (3,2) (4,4) (6,1)
U_3	(3,6) (4,4) (6,3)
U_4	[∅,2,∅][1,4,∅][1,4,∅][2,4,1] (3,2) (6,1) (6,6) (7,5)
U_5	(7,8)

(a) Auxiliary Lists



(b) Query Binary Tree

Fig. 2. Query processing based on FC

Query Processing. We answer a query Q by top-down traversing the tree nodes and processing their associated U_i lists. We first search the anchor point of Q in U_{root} using standard binary search. Then according to the dominance relationship between Q and L_{root}, we search U_{left} or U_{right} following the pointers in U_{root}, until we determine $maxRS(Q)$ or reach a leaf. In other words, the use of Algorithm 1 is now replaced with a constant cost operation (of point-wise dominance checking), thanks to FC.

An Example. Let Q be $(2, 5)$ and we use the data structure in Fig. 2. Anchor value of Q is 2, and binary search in the root node (i.e., U_4) results in $\emptyset$, indicating no point in U_4 dominates Q, hence $Q \in LOW(L_4)$. We need to continue to search query's anchor point in the left child node (i.e., U_2); this can be done by following the pointer stored in $U_4[1]$ (the last entry we searched in the parent) to go to $U_2[2]$ directly. Since we only sampled even entry of U_2 in U_4, even though $U_2[2]$ is not the anchor point of Q, we still need to check if the previous entry $U_2[1]$ is. The answer is true, and as $U_2[1]$ dominates Q, we get $Q \in HIGH(L_2)$ based on Lemma 6. The next step is to search Q's anchor point in U_3 by following pointer to U_3 attached with $U_2[1]$, and we get $\emptyset$. This indicates no dominate point of Q in U_3, so $Q \in LOW(L_3)$. Now, we can decide $maxRS(Q) = 2$, and the search stops.

Analysis of Space Complexity. We have n L_k lists. Denote the length of each list L_k as l_k, and the total CP lists size is $S = \sum_{k=1}^{n} l_k$. Let leaf node has level 1, and the level of the root is h. Denote S_i (resp. W_i) as the total size of L (resp. U) lists associated in tree level i. Then we have $\sum_{i=1}^{h} S_i = \sum_{k=1}^{n} l_k = S$, and $W_i = S_i + \frac{W_{i-1}}{2}$. Thus, the total size of all U lists is $\sum_{i=1}^{h} W_i = S_1 + (\frac{1}{2}S_1 + S_2) + \cdots + (\frac{1}{2^{h-1}}S_1 + \frac{1}{2^{h-2}}S_2 + \cdots + S_h)) < 2(S_1 + S_2 + \cdots + S_h) = 2S$. Therefore, the total space of U_i and L_i lists is linear in total number of CPs.

Analysis of Time Complexity. The size of U_{root} list is $W_h = \frac{1}{2^{h-1}}S_1 + \frac{1}{2^{h-2}}S_2 + \cdots + S_h < S$. The binary search in U_{root} has cost $O(\log S) = O(\log n)$. We inspect at most $\log n$ non-root nodes in the tree, and each with constant cost $O(1)$. Therefore, the total search cost is $O(\log n + \log n) = O(\log n)$.

5 Index Construction

In this section, we introduce the index construction and complexity analysis.

If we directly apply existing method in [2] to compute the index, for each combination of (a_i, b_i), we initiate a query taking $O(n \log n)$ to find its maxRS value, and finally obtain all the CPs in time $O(n^5 \log n)$. Whereas in the next subsection, we present an algorithm that constructs the index in time $O(n^3 \log n)$.

5.1 Main Idea

Given a CP in cell $M[i, j]$, it means there exists a rectangle r of size $a_i \times b_j$ that contains $M[i, j]$ number of points from P. We refer such a rectangle r as the rectangle represented by this CP. Then we have the following lemma:

Lemma 7. *The rectangle represented by each changing point must touch points from P on all its four edges.*[2]

Lemma 7 narrows down our search space into $O(n^4)$ rectangles to find all the CPs. An implied property of CP is that among all the rectangles containing k points from P, the rectangle represented by a CP in L_k is the one that cannot be dominated[3] by any other rectangles. This can be clearly observed from the matrix in Fig. 1(c). Thus, instead of generating all the $O(n^4)$ rectangles whose four edges touch points in P, we find L_k $(k \in [2 \dots n])$[4] in the following way: for each possible rectangle width a_i, we find the rectangle r_{min} with minimum height (i.e. min b_i) such that the pair of points (p_l, p_r) which forms interval a_i lies on the left and right vertical edges of r_{min} respectively, and r_{min} contains exactly k points. Then r_{min} is a candidate CP in L_k. After all the a_i are processed, we get $O(n^2)$ r_{min} in L_k, and eliminate the ones that are dominated by others, we get all the CPs in L_k.

In this idea, we generate $O(n^3)$ candidate rectangles. The total construction cost is closely related to the cost of finding r_{min} for each a_i and k value. We introduce the structure to find r_{min} next.

Y-coordinate Distance Arrays. Given a_i, recall that points in S_{a_i} are sorted according to y-coordinates, we assume the size of S_{a_i} is $|S_{a_i}|$, and denote $S_{a_i} = \{p_1, p_2, \dots, p_{|S_{a_i}|}\}$. We construct $|S_{a_i}|-1$ number of y-coordinate distance arrays over points in S_{a_i} as the following:

- $D_2 : \{|p_1.y - p_2.y|, |p_2.y - p_3.y|, |p_3.y - p_4.y|, \cdots, |p_{|S_{a_i}|-1}.y - p_{|S_{a_i}|}.y|\}$
- $D_3 : \{|p_1.y - p_3.y|, |p_2.y - p_4.y|, \cdots, |p_{|S_{a_i}|-2}.y - p_{|S_{a_i}|}.y|\}$
- $\cdots$
- $D_{|S_{a_i}|} : \{|p_1.y - p_{|S_{a_i}|}.y|\}$

[2] The proof is obvious and omitted due to space limitation.

[3] $R1$ is dominated by $R2$ if both the width and height of $R1$ are larger than or equal to that of $R2$, the dominance relationship is formally defined in Definition 3.

[4] L_1 contains the only point $(0, 0)$.

Assume the two points forming a_i are p_l and p_r $(l < r)$ in S_{a_i}, then for each value $k \in [2, |S_{a_i}|]$, we find the height of the minimum rectangle that contains k points by calling $RMQ_{D_k}(r-k+1, l)$ on D_k. The index of D_k starts from 1. Now the construction cost depends on the cost of answering Range Minimum Queries on D_k.

Update Distance Arrays. While techniques for answering RMQ on 1D array have been extensively studied in current literatures, and the most efficient ones achieve O(1) query time with linear auxiliary index space, they work mainly on static underlying arrays. If adapted to our dynamic distance arrays D_k, which change along with a_i, the cost of updating the auxiliary index structure for RMQ outweighs the cost saved by using it.

Another challenge is that after processing interval a_i, and move to the next closest one, say a_{i+1}, such that S_{a_i} and $S_{a_{i+1}}$ differ by only 1 point p_j, then for each D_k, the number of distance values affected by p_j is $O(k)$. For example, if p_3 is removed from D_3, the entries being affected are $|p_1.y - p_3.y|$, $|p_2.y - p_4.y|$, and $|p_3.y - p_5.y|$. Thus, the total number of entries need to be updated in all D_k $(k \in [2 \ldots |S_{a_i}|])$ is $O(n^2)$ for each a_i, and totally $O(n^4)$ for all a_i $(i \in [1 \ldots |A|])$.

To overcome the above challenges, we devise a tree-based structure that supports RMQ in time $O(\log n)$ as well as updating of each D_k in $O(\log n)$ according to the following observation.

Lemma 8. *When a point p is removed from list S_{a_i}, the $O(k)$ entries that have to be updated in D_k are continuous, and the updated results are identical to a continuous part of entries in D_{k+1}.*

Left-Complete Binary Tree. We maintain a left-complete binary tree T_k over each list D_k in the sense that the left subtree of each internal node is complete. The leaves of T_k store the elements of D_k, and internal nodes correspond to ranges of consecutive elements of the list. Each internal node v stores a pointer to a leaf μ in the subtree rooted at v with minimum distance value. At any time, the size of the data structure is linear in the number of elements present in the list. The key value of each node is set as the following: (1) Leaf node's key is the index of the element in D_k it stores. (2) The i^{th} internal node in level l has key value $2^l * (i-1) + 2^{(l-1)} + 0.5$ (l starts from 0, i starts from 1, and leaf has level 0).

When a point is removed from S_{a_i}, and $O(k)$ continuous entries in the range $D_k[l \ldots r]$ need to be updated, we firstly search l and r in T_k and T_{k+1} simultaneously. Denote the path of searching l (resp. r) in T_k as FL_{T_k} (resp. FR_{T_k}), and the forest contained between FL_{T_k} and FR_{T_k} as F_k. Updating $D_k[l \ldots r]$ is achieved by moving F_{k+1} to the corresponding position of F_k. The above tree structure and node key settings guarantee all the key values appear in $FL_{T_{k+1}}$ and $FR_{T_{k+1}}$ also appear in FL_{T_k} and FR_{T_k}. Thus the moving can be done by simply adjusting pointers of nodes in FL_{T_k}, FR_{T_k}, and root nodes in forest F_{k+1}. After the moving step, a bottom-up traversing of paths FL_{T_k} and FR_{T_k} is performed to update the min pointers in each node. Therefore, for each

a_i, although we need to update $O(n^2)$ distance values, the total update cost is $O(n \log n)$. The RMQ can be solved in $O(\log n)$ time as usual.

5.2 The Complete Algorithm

Algorithm 2 shows our index construction method.

1. We initialize n number of empty sorted lists (Line 1). Each list L_k stores the currently generated candidate rectangles containing k points. When a new rectangle R needs to be inserted into L_k(Lines 13–14), we check if R is dominated by existing entries in L_k. If not, we insert R into L_k, and remove all elements in L_k that are dominated by R. Here, a rectangle R is represented by a pair of value (a, b), where a refers to width and b refers to height.
2. At the beginning, we build the $n-1$ y-coordinate distance arrays for all points in P, and the auxiliary trees for fast RMQ and range update (Lines 4–7).
3. For each x-interval a_i, and for each $k \in [2 \dots |S_{a_i}|]$, we query $b_{min} = RMQ_{T_k}(r - k + 1, l)$[5] on tree T_k to find the minimum height of rectangle containing k points, l and r are the indexes of the two points in S_{a_i} that form the x-interval a_i. Then we add (a_i, b_{min}) as a candidate rectangle to L_k (Lines 10–14). For each a_i, after all the possible k values are processed, the auxiliary trees are updated as described in Sect. 5.1 (Lines 15–16).

Total Cost Analysis. There are totally $O(n^3)$ rectangles generated, each takes $O(1)$, and checking whether it is dominated by existing candidates takes $O(\log n)$, so in total it is $O(n^3 \log n)$ time, and $O(n^3)$ space to store rectangles. The first construction of y-coordinate distance arrays and auxiliary trees costs $O(n^2)$ in time and $O(n^2)$ in space. For each change of x-interval a_i, $O(n^2)$ distance values are updated from all trees, which costs only $O(n \log n)$ in time as analyzed before. There are $O(n^2)$ x-intervals, so total tree update cost is $O(n^3 \log n)$.

Therefore, our index can be constructed in $O(n^3 \log n)$ time, and $O(n^3)$ space.

6 Other Problems

In this section, we present the applications and advantages of our method for solving other classic computational geometry problems.

General Weighted maxRS Problem and Approximate Version. If points in P have different weights, the only difference with analysis in previous sections is that the number of different maxRS values cannot be bounded by $O(n)$. In this case, the number of CPs is upper bounded by $O(n^4)$. The construction can be done in $O(n^4)$ by simply finding all the rectangles whose four edges touch

[5] In the pseudocode of Algorithm 2 Line 12, we make the optimization by calling $RMQ_{T_k}(1, |T_k|)$ rather than $RMQ_{T_k}(r - k + 1, l)$. We omit the proof here.

Algorithm 2. ConstructIndex(list P)

Input : data point list P
Output: The list of changing points L_k for each $k \in [2, n]$

```
1  L_k ← ∅, ∀k ∈ [2, n] ;
2  P^y ← P sorted in decreasing order of y-coordinates ;
3  P^x ← P sorted in increasing order of x-coordinates ;
4  for k ← 2 to n do
5      D_k ← { |p_i^y.y − p_{i+k−1}^y.y| | i ∈ [0, n − k + 1] };  /* build dist arrays */;
6      T_k ← the binary tree constructed from D_k ;
7      T'_k ← a copy of T_k ;
8  for i ← 0 to n − 2 do
9      for j ← n − 1 downto i + 1 do
10         a_i ← p_j^x.x − p_i^x.x;  /* Process each of the O(n^2) x-intervals */;
11         for k ← 2 to j − i + 1 do
12             (b_min, p_c^y, p_d^y) ← RMQ(1, |T_k|) ;
13             if (p_c^y, p_d^y) covers (p_i^x, p_j^x) then
14                 L_k.addresult(a_i, b_min);  /* add result and remove dominance
                   */;
15         for k ← 2 to j − i + 1 do
16             updatetree(T_k, p_j^x); /* update entries affacted by p_j^x in T_k */;
17     for k ← 2 to n − i + 1 do
18         updatetree(T'_k, p_i^x) ;
19         T_k ← T'_k;                          /* restore T_k */;
20 return { L_k | k ∈ [2, n] };
```

points and finally get all the CPs. Despite the increase of indexing cost, the query time remains the same as $O(\log n)$.

To answer $1 - \epsilon$ approximate maxRS queries with 100 % guarantee, the only change is to include only the i-lines in the index, where $i = \{1, c, c^2, \dots\}$, where $c = \frac{1}{1-\epsilon}$. When c is a constant, this reduces the index size down to $O(n^2 \log n)$ with the same $O(\log n)$ query time complexity (See Table 1).

k-Enclosing Problem. K-enclosing rectangle problem [8–11], is the problem of finding the minimum axis-parallel rectangle that contains exact or at least k points from P, in terms of measurement like rectangle area or perimeter.

The most efficient existing method [11] solves this problem in $O(n + k(n - k)^2) = O(n^3)$ when $k = \frac{n}{2}$. Using our index, we solve the problem in $O(1)$ if each L_k is sorted according the query measurement, or $O(n)$ by linear scan L_k if L_k is not sorted. Both cases are way better than existing time bounds achieved. Even without index, our method can be modified to answer k-enclosing problem in time $O((n - k)^2 k + n \log n) = O(n^3)$ when $k = \frac{n}{2}$, which meets the bound of the best existing method.

Furthermore, our method addresses k-enclosing problem for all possible values of k in time $O(n^3 \log n)$ including both indexing and query time. While existing method takes $O(n^4)$ in total in order to give answer for all possible values of k.

Maximum Point Enclosing Problem. Maximum point enclosing problem can be seen as the inverse problem of k-enclosing problem. It aims at finding the maximum number of points a rectangle can enclose, and the rectangle measurement (area or perimeter) is no larger than given query value Q.

Existing method [2] can be adapted to solve the above problem in $O(n^3 \log n)$ time by trying for each possible width a_i ($O(n^2)$ number of possibilities), together with the largest height b_i computed so that the rectangle size is within given query parameter, finding the maxRS value of (a_i, b_i) in time $O(n \log n)$, and the global maximum maxRS value is the final answer.

Whereas our method solves the problem in $O(\log^2 n)$ by a nested binary search procedure[6] similar as the one presented in Sect. 4 if L_k lists are sorted on query measurement, or $O(n \log n)$ if L_k lists are not sorted. Even if indexing time is considered, we take $O(n^3 \log n)$ total time, but be able to answer any given query parameter efficiently, while existing method sloves each query using $O(n^3 \log n)$ time.

7 Experiment

In this section, we perform empirical experiments to confirm our theoretical analysis of algorithms' performance and demonstrate the substantial query performance improvement by our proposed method in practice.

Experiment Setting. We perform experiments on both synthetic and real datasets. The synthetic dataset contains 10,000 points generated with uniform distribution. The ranges of x and y coordinates are both set to $[0, 10^5]$. The real dataset is drawn from the publicly available NorthEast (NE) dataset[7], which contains 20,000 postal addresses in New York, Philadelphia and Boston.

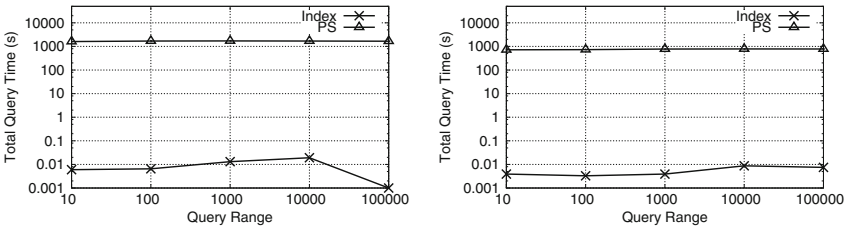
We compare our method with the plane-sweep algorithm [2] (denoted as **PS**), as it is the most efficient one for the exact maxRS problem; our index-based algorithm is denoted as **Index**. Both methods are implemented in C++, and experiments are conducted on a PC with Intel Core i7 CPU 2.7 GHz with 8 GB of memory.

Varying Query Rectangle Size. We test the performance of both methods on various query rectangle sizes. The five groups of rectangle size ranges are $[0^2, 10^2]$, $[10^2, 100^2]$, $[100^2, 1000^2]$, $[1000^2, 10000^2]$, and $[10000^2, 100000^2]$.

[6] Based on the property that $\forall q \in L_k$, there exists $q' \in L_{k-1}$ such that q' dominates q.
[7] www.rtreeportal.org.

We produce 1,000 queries with sizes generated uniformly from each group. We show the *total query time* for each group in Fig. 3.

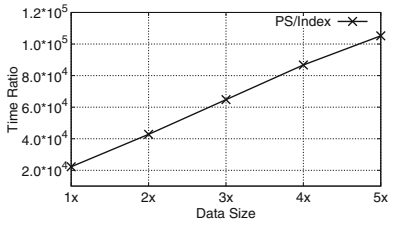
It can be seen that our method is around 5 orders of magnitude faster than existing method PS. This demonstrates the substantial advantage of our index-based method to efficiently support batch query workloads. When query range increases, the running time of PS remains steadily due to the nature of plane-sweeping, while our method takes slightly more time when query size is in middle range. This is because the size of k-lines with k around $n/2$ tends to be larger than other values — as can be seen in Fig. 1(c).



(a) NE (b) Random

Fig. 3. Vary query range

Vary Dataset Size. We start with a synthetic dataset with 2000 points, and increase its size till 5x, and measure the ratio between PS's total query time over that of Index's. The result is plotted in Fig. 4(a). Clearly, our method has better scalability than PS as the time ratio raises up quickly when the data size increases. This is expected as our query time increases only logarithmically with n, while PS's increases superlinearly.



(a) Vary Data Size

n	$CPNum$	$CPNum/n^2$	$size$
m	c	$\frac{c}{m^2}$	0.4KB
$10m$	$97c$	$0.97 \cdot \frac{c}{m^2}$	35KB
$100m$	$7597c$	$0.75 \cdot \frac{c}{m^2}$	2.7MB
$1000m$	$731026c$	$0.73 \cdot \frac{c}{m^2}$	314MB

(b) Index Size

Fig. 4. Other experiments

Index Size. We start with an initial dataset size of $m = 10$ which results in total number of changing points $c = 36$. Then we increase the dataset till 1000 fold and show the increase of CPs in Fig. 4(b). As mentioned in Sect. 3.2, although our current bound on our index size is $O(n^3)$, we conjecture that it could be $O(n^2)$. Hence, we also show in the third column the ratio of index size over square of data size. The result strongly suggests that the conjecture may be true. We show the actual index size in the last column.

8 Conclusion

In this paper, we study the maximizing range sum problem. Existing methods for both exact and approximate query processing require $\Omega(n \log \log n)$ time. We propose a novel method based on indexing changing points, which results in an index of size $\Theta(n\lambda) = O(n^3)$. The index enables us to devise a non-trivial query processing algorithm with $O(\log n)$ complexity. Our method provides new space-time tradeoff for the maxRS and related problems. Experiments on real and synthetic datasets verify the efficiency of our method.

Acknowledgements. This work was partially done when X. Zhou and W. Wang visited Hong Kong Baptist University. W. Wang was supported by ARC DP 130103401 and 130103405. J. Xu was supported by HK-RGC grants 12201615 and HKBU12202414.

References

1. Imai, H., Asano, T.: Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. J. Algorithms **4**(4), 310–323 (1983)
2. Nandy, S.C., Bhattacharya, B.B.: A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. Math. Appl. **29**(8), 45–61 (1995)
3. Choi, D.W., Chung, C.W., Tao, Y.: A scalable algorithm for maximizing range sum in spatial databases. Proc. VLDB Endow. **5**(11), 1088–1099 (2012)
4. Tao, Y., Hu, X., Choi, D.W., Chung, C.W.: Approximate MaxRS in spatial databases. PVLDB **6**(13), 1546–1557 (2013)
5. Choi, D.W., Chung, C.W., Tao, Y.: Maximizing range sum in external memory. ACM Trans. Database Syst. **39**(3), 21:1–21:44 (2014)
6. Das, A.S., Gupta, P., Srinathan, K., Kothapalli, K.: Finding maximum density axes parallel regions for weighted point sets. In: CCCG (2011)
7. Goodrich, M.T., Tsay, J.-J., Vengroff, D.E., Vitter, J.S.: External-memory computational geometry (preliminary version). In: FOCS, pp. 714–723 (1993)
8. Aggarwal, A., Imai, H., Katoh, N., Suri, S.: Finding k points with minimum diameter and related problems. J. Algorithms **12**, 38–56 (1991)
9. Datta, A., Lenhof, H.E., Schwarz, C., Smid, M.: Static and dynamic algorithms for k-point clustering problems. J. Algorithms **19**, 474–503 (1995)
10. Eppstein, D., Erickson, J.: Iterated nearest neighbors and finding minimal polytopes. Discrete Comput. Geom. **11**, 321–350 (1994)
11. Segal, M., Kedem, K.: Enclosing k points in the smallest axis parallel rectangle. Inf. Process. Lett. **65**, 95–99 (1998)
12. Matouek, J.: On geometric optimization with few violated constraints. Discrete Comput. Geom. **14**, 365–384 (1995)
13. Hartigan, J.A.: Clustering Algorithms. Wiley, New York (1975)
14. Abellanas, M., Hurtado, F., Icking, C., Klein, R., Langetepe, E., Ma, L., Palop, B., Sacristán, V.: Smallest color-spanning objects. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 278–289. Springer, Heidelberg (2001)
15. Efrat, A., Sharir, M., Ziv, A.: Computing the smallest k-enclosing circle and related problems. Comput. Geom. Theory App. **4**(3), 119–136 (1994)

16. Mahapatra, P.R.S., Karmakar, A., Das, S., Goswami, P.P.: k-enclosing axis-parallel square. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part III. LNCS, vol. 6784, pp. 84–93. Springer, Heidelberg (2011)
17. Das, S., Goswami, P.P., Nandy, S.C.: Smallest k-point enclosing rectangle and square of arbitrary orientation. Inf. Process. Lett. **94**, 259–266 (2005)
18. Mukherjee, M., Chakraborty, K.: A polynomial time optimization algorithm for a rectilinear partitioning problem with applications in VLSI design automation. Inf. Process. Lett. **83**, 41–48 (2002)
19. Tiwari, S., Kaushik, H.: Extracting region of interest (ROI) details using LBS infrastructure and web databases. In: MDM 2012, pp. 376–379 (2012)

http://www.springer.com/978-3-319-44402-4

Database and Expert Systems Applications
27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part I
Hartmann, S.; Ma, H. (Eds.)
2016, XXVIII, 449 p. 155 illus., Softcover
ISBN: 978-3-319-44402-4