Planar Convex Hull Range Query and Related Problems

Nadeem Moidu * Jatin Agarwal † Kishore Kothapalli ‡

Abstract

We consider the planar convex hull range query problem. Let P be a set of points in the plane. We preprocess these points into a data structure such that given an orthogonal range query, we can report the convex hull of the points in the range in $O(\log^2 n + h)$ time, where h is the size of the output. The data structure uses $O(n \log n)$ space. This improves the previous bound of $O(\log^5 n + h)$ time and $O(n \log^2 n)$ space. Given a range query, it also supports extreme points in a given direction, tangent queries through a given point, and line-hull intersection queries on the points in the range in time $O(\log^2 n)$ for each orthogonal query and $O(\log n)$ time for each additional query on that range. These problems have not been studied before.

1 Introduction

Planar convex hull is a well studied topic in computational geometry. Let P be a set of n points on the plane. Overmars and van Leeuwen gave a data structure which allows insertions and deletions in P in $O(\log^2 n)$ time and reporting of points on the hull in $O(\log n + h)$ time, where h is the number of points on the hull [7]. Instead of supporting reporting of the entire hull, recent works provide data structures to support common queries on the convex hull, $CH(P)$, without actually computing it. The following queries are typically studied:

1. *Extreme point query*: Find the most extreme vertex in $CH(P)$ along a query direction
2. *Tangent query*: Find the two vertices of $CH(P)$ that form tangents with a query point outside the hull
3. *Line stabbing query*: Find the intersection of $CH(P)$ with an arbitrary query line

These queries can be supported in $O(\log n)$ time by the structure of Overmars and van Leeuwen [7]. Brodal and Jacob gave a solution which supports insertions and deletions in $O(\log n)$ amortized time and the first two queries in $O(\log n)$ time without actually computing the hull [3]. Chan gave a data structure which supports the third query in $O(\log^{3/2} n)$ time [4] and later improved it to $O(\log^{1+\epsilon} n)$ time [6].

In this paper, we study the orthogonal range query versions of the above problems. Given an orthogonal range query of the form $q = [x_{low}, x_{high}] \times [y_{low}, y_{high}]$, we support the above queries for the points in $P \cap q$. Brass et al. first gave a solution to report the convex hull of an orthogonal range in $O(\log^5 n + h)$ time in [2]. The other problems are being studied for the first time but the data structure in [2] can be enhanced to support these queries in $O(\log^5 n)$ time per orthogonal range query and an additional $O(\log^2 n)$ time for any of the above queries. Our data structure takes $O(\log^2 n)$ time to process one orthogonal range query. Once this is done, we can report the points on the hull of $P \cap q$ in $O(h)$ time and any of the above queries in $O(\log n)$ time.

2 Overview

In a standard two dimensional range tree, a query is divided vertically into $O(\log n)$ rectangular regions where each region corresponds to a canonical node in the primary tree. Each of these primary regions are further divided horizontally into $O(\log n)$ regions corresponding to canonical nodes in the secondary tree. However, these horizontally divided regions are independent of each other, i.e. they correspond to different intervals in different primary regions. In our data structure we modify the secondary trees such that, for a given query, the horizontal divisions are same across all canonical primary node regions. So an orthogonal query is divided into a grid of $O(\log n) \times O(\log n)$ regions which are perfectly aligned as shown in figure 1. By having the divisions aligned like this, we are able to discard a large number of regions which do not contribute to the final hull without processing them. This idea is similar to the method used by Abam et al. to enhance kinetic kd-trees in [1].

3 Data Structure

Let P be a static set of points on a plane. We construct a one dimensional range tree, T_y of all the points based on their y coordinates. We call this the *template* tree. Given a subset S of the point set P, the *contracted* tree of T_y with respect to S is defined as the tree obtained

*nadeem.moiduug08@students.iiit.ac.in
†jatin.agarwal@research.iiit.ac.in
‡kkishore@iiit.ac.in

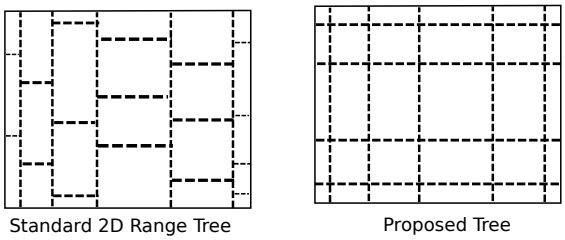

Figure 1: Example of how a query is split into canonical node regions for (a) normal 2D range tree and (b) our tree

by removing all subtrees which do not have a leaf in S and contracting all nodes which have only one child. See Figure 2. Since a contracted tree is a full binary tree (i.e. a tree in which each node has exactly either zero or two children), it takes $O(|S|)$ space.

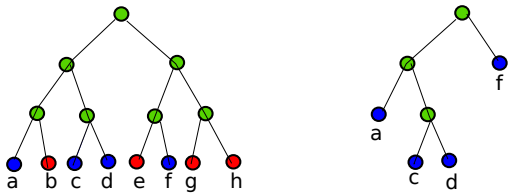

Figure 2: Original tree and the contracted tree for the set {a,c,d,f}

Next, we construct a primary tree, T, which is a one dimensional range tree based on the x coordinates. Each node in this primary tree contains a secondary tree which is a contracted version of T_y with respect to the points in the corresponding x range.

A convex hull can be divided into four parts based on the extreme points along each axis. The upper right part goes from the point with maximum y coordinate to the point with maximum x coordinate. The other parts are similar. Our data structure is designed to compute the upper right part of the convex hull. The other parts can be computed similarly and the four parts can be joined together to obtain the final convex hull. From here on, we will refer to the upper right convex hull as *urc*-hull.

We now describe the information stored at each secondary tree node. Each internal secondary tree node corresponds to a set of points which is a union of two disjoint sets of points, separated by a horizontal line. So the *urc*-hull of the points in a node will comprise a part of each of the child node *urc*-hulls and the outer common tangent (bridge) connecting them. We store the following variables in each internal node, u:

1. A boolean variable which indicates whether the left child (horizontally lower part) contributes any part to the *urc*-hull. If this variable is false, then the next three parameters are set to $NULL$.
2. The outer common tangent (bridge) connecting the *urc*-hulls of the children, represented by the points where it intersects the *urc*-hulls, $B_l(u)$ and $B_r(u)$.
3. Both neighbors of $B_l(u)$ and $B_r(u)$ in the *urc*-hulls of the respective child. (These parameters are required for computing the common tangent between two hulls).
4. Indices of $B_l(u)$ and $B_r(u)$ in the *urc*-hulls of the respective child, $Index_l(u)$ and $Index_r(u)$. We need these two parameters to know the portion of each child *urc*-hull that contributes to the *urc*-hull of the current node.
5. The y range spanned by the node.
6. Number of points in the *urc*-hull, $N(u)$.

For leaf nodes, we simply store the point. Since each secondary tree node stores a constant amount of information, the space taken by each primary tree node is proportional to the number of points in the interval. So the overall space taken is $O(n \log n)$.

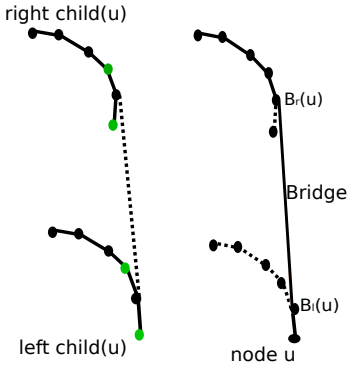

Figure 3: Different parameters stored at each node. The neighbors of $B_l(u)$ and $B_r(u)$ are marked green.

Note that we are not storing the *urc*-hull, as it is, in each node. However, given indices, i and j, we can report the points in the *urc*-hull from i to j in $O(\log n + j - i + 1)$ time as shown in section 5.

4 Preprocessing

The primary tree is constructed as a standard one dimensional range tree based on x coordinates. We pro-

cess the primary tree from top to bottom to obtain the secondary trees (contracted trees) at each primary tree node. The interval at the root node includes all the points in P, so the template tree, T_y, is stored as it is. Note that the interval corresponding to a non-root primary node is a subset of the interval corresponding to its parent. For each child, we replicate the tree present in its parent and then contract it with respect to the points in the interval of the node. A tree can be contracted with respect to a set of points by removing all leaves not present in the child and then updating the parents of the removed leaves as required. Each node in the tree is processed a constant number of times, so the time taken for this stage is $O(n \log n)$.

Once the tree structure is completed, we start storing the required information from bottom to top starting from the leaves. Except the common bridge, all the other parameters of the node can be easily found in constant time. The bridges can be computed as follows: At each node, discard the points of the child hulls which do not form part of the parent hull. Since each each point is discarded atmost once, it takes amortized constant time per point. So the time taken for preprocessing is $O(n \log n)$ and the space usage is $O(n \log n)$.

5 Query Algorithm

Given an orthogonal query $[x_{low}, x_{high}] \times [y_{low}, y_{high}]$, we can identify $O(\log n)$ canonical nodes corresponding to the y range, $[y_{low}, y_{high}]$ in the template tree T_y. We will also identify the $O(\log n)$ canonical nodes corresponding to the x range in the primary tree. We then find out the nodes corresponding to the T_y nodes in each of the primary tree nodes. There are three cases here:

1. The node present in T_y exists in the contracted tree as it is. In this case, we simply use that node.

2. The node was removed because both its children were removed. This means that the node was empty, so this node does not contribute any point to the *urc*-hull.

3. The node was removed because it was the only child of its parent. In this case, we check the node present in its place to get the required information, if any.

So an orthogonal range query gets split into $O(\log n) \times O(\log n)$ secondary tree nodes, which are well aligned as shown in figure 1. These cases can be identified while doing a normal one dimensional range tree query on the secondary nodes.

Lemma 1 *The upper right convex hull of the orthogonal range can be computed in $O(\log^2 n)$ time.*

Proof. We define the region covered by each of the $O(\log n) \times O(\log n)$ secondary tree nodes as a *block*. Start by identifying the non empty *block*s. If a *block* is non empty, then no *block* which is to its left and bottom can contribute points to the *urc*-hull. A *block* is called a *candidate block* if it is non-empty and all *block*s to the right and top of its top right corner are empty. See figure 4. Based on this observation, the *candidate block*s can be identified as follows: start from the bottommost non-empty *block* in the rightmost column. This is a *candidate block*. If there exists at least one non empty *block* above it in the same column, move to the next (non-empty) *block* in the upwards direction. Otherwise, move one *block* to the left. Continue this till we reach the top left *block*. Every *block* visited in this process is a *candidate block*. Since we are moving only up or left, we will move in the left direction at most $O(\log n)$ times and in the up direction at most $O(\log n)$ times. So the total number of *candidate block*s is at most $O(\log n)$.

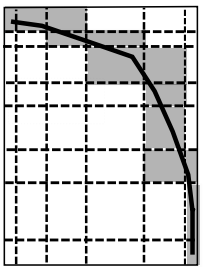

Figure 4: Example of a query. The candidate blocks have been darkened.

Now process the candidate blocks in the order they were visited. The individual *urc*-hulls of each node can be merged together to form the complete *urc*-hull in a manner similar to Graham's scan. This method has been used before, e.g. see [5]. Maintain the *urc*-hull up to the current block in a stack, H. Each block, contributes atmost one continuous range to the *urc*-hull. Each element of the stack contains a pointer to a candidate block, $H(u)$ and the indices of the start and end points of this range, $H_s(u)$ and $H_e(u)$. First, push the right bottom block. Process each subsequent block, v as follows: Compute the common tangent between the current block, v and the *urc*-hull on the top of the stack, $top(H)$. If this tangent does not intersect $top(H)$ between the range $top(H_s)$ and $top(H_e)$, then the current *top* does not contribute to the *urc*-hull anymore. So pop out *top* from the stack and compute the tangent with the new top of the stack. Continue this till the top of the stack is not popped out. Now push the current block to the top of the stack and update the ranges appropriately based on the tangent information.

The time taken is mostly for computing the tangents. This has to be done exactly once for each time a *urc*-hull is pushed or popped. Each tangent computation using the method of Overmars and van Leeuwen [7] takes $O(\log n)$ time. This method compares a point on each of the hulls and discards either the portion before it or the portion after it in the corresponding hull. This is where we use the neighbors of the bridges stored in each secondary tree node. This tangent computation is done $O(\log n)$ times. So the overall time complexity is $O(\log^2 n)$. □

The above merging algorithm returns a stack of the secondary tree nodes and the indices of the range that each of these nodes contribute to the *urc*-hull. The line segment connecting the end points of two adjacent nodes in this stack gives the bridge connecting them. Using this structure, we can report all the points on the *urc*-hull in $O(\log n + h)$ time as shown in algorithm 1.

Input: Tree node u, Indices i and j
Output: Points on the *urc*-hull corresponding to u from indices i to j, inclusive
if *u is a leaf node* **then**
  Report the point if $i = j = 1$
else
  if $Index_l(u) > i$ **then**
    Report points in the range of the left subtree which forms part of the range $[i, j]$ in the parent hull
  end
  if $i \leq Index_l(u) \leq j$ **then**
    Report $B_l(u)$
  end
  if $i \leq Index_l(u) + 1 \leq j$ **then**
    Report $B_r(u)$
  end
  if $Index_l(u) + 1 < j$ **then**
    Report points in the range of the right subtree which forms part of the range $[i, j]$ in the parent hull
  end
end

Algorithm 1: Algorithm to report the points on the *urc*-hull of a node from given indices i to j, inclusive

5.1 Other Problems

We now explain how to solve the extreme point query, line stabbing query and tangent query problems using the stack obtained above without constructing the entire convex hull. Recall that the convex hull was divided into four parts based on the extreme points. For the problems discussed in this section, the answer could be in any of these four parts. It is easy to identify the exact part(s) by comparing the extreme points with the query parameter.

The basic idea is the following: By comparing an edge of the hull with a query parameter, we can discard, in constant time, either the part before the edge or the part after the edge. We proceed as follows: Compare each bridge connecting adjacent elements in the stack with the query parameter. If the required output lies on one of the bridges, then we report the answer and stop. Otherwise, this will help in identifying the exact node which contains the required output. This takes time proportional to the number of bridges, which is $O(\log n)$. Once the node is identified, by comparing the bridge at the root of the node with the query parameter, we can decide whether we should take the root, or go to the left or right subtree. This takes time proportional to the height of the tree which is also $O(\log n)$. So the overall time taken is $O(\log n)$.

For line stabbing query, there are at most two points to be reported. We have to query for each of them separately. Otherwise, it will not be possible to discard a half at each stage in the above algorithm.

6 Future Work

It might be possible to improve the bounds given in this paper. A more interesting problem is to make the set of points dynamic by allowing insertions and/or deletions. The problem is also open in higher dimensions.

The modified range tree approach can be used to improve various range query problems like reporting the smallest enclosing disk or the width of the points in a query rectangle.

References

[1] M. A. Abam, M. de. Berg and B. Speckmann, Kinetic kd-Trees and Longest-Side kd-Trees. In SICOMP 39:1219-1232, 2009.

[2] P. Brass, C. Knauer, C. S. Shin, M. Schmid and I. Vigan. Range-Aggregate Queries for Geometric Extent Problems. In Proc. 19th Computing: Australasian Theory Sympos. CATS 141:3-10, 2013.

[3] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In Proc. 43rd IEEE Sympos. Found. Comput. Sci., pages 617-626, 2002.

[4] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In J. ACM, 48:1-12, 2001.

[5] T. M. Chan. and E. Y. Chen, Multi-Pass Geometric Algorithms. In Discrete and Comput. Geom.,37(1): 79-102, 2007.

[6] T. M. Chan. Three Problems about Dynamic Convex Hulls. In Proc. 27th ACM Sympos. Comput. Geom. 27-36, 2011.

[7] M. H. Overmars and J. van Leeuwen. Maintenance of Configurations in the Plane. In J. Comput. Syst. Sci. 23:166-204, 1981.