Static and Dynamic Convex Hull Algorithms

Chongshan Chen

CS 633

December 2007

Static and Dynamic Convex Hull Algorithms

Abstract

I survey the existing static convex hull algorithms (Graham's Scan, Jarvis' March, and Chan's optimal output-sensitive algorithm) and Overmars' dynamic convex hull algorithm and their data structures in the planar case, and compare their computational complexities. My work has a twofold purpose: it introduces the area to the non-specialist and reviews the state-of-the-art algorithms for the specialist.

1 Introduction

Finding the convex hull of a set of points is the most elementary interesting problem in computational geometry. It arises because the hull quickly captures a rough idea of the shape or extent of a data set. Convex hull also serves as a first preprocessing step to many, if not most, geometric algorithms. It finds use in a number of different applications including: collision avoidance in robotics, pattern recognition and digital image processing. Also, the fundamental process of finding a convex hull is often embedded inside other algorithms that themselves are important in a range of applications. Such 'next tier' algorithms include: determining whether a point lies 'amongst' a set of other points; determining whether two sets of points 'overlap'; finding the smallest rectangular box that will encompass a set of points; and finding a 'rough' description of the shape defined by a series of points. This motivates me to study a wide variety of different convex hull algorithms, which lead to interesting or optimal solutions.

In this paper, I studied several static convex hull algorithms, including Graham's Scan Algorithm which has O(nlogn) time complexity, Jarvis' March Algorithm which has O(nh) time complexity, and Chan's optimal output-sensitive algorithm which has O(nlogh) time complexity, where n is the number of points in the plane, and h is the number of points on the convex hull. Furthermore, I researched on the Overmar and van Leeuwen's dynamic convex hull algorithm which processes the insertion and deletion of points in $O(\log^2 n)$ time.

2 Graham's Scan

The Graham's scan is a method of computing the convex hull of a given set of points in the plane with time complexity O(n log n). It is named after Ronald Graham, who published the original algorithm in 1972[2]. The algorithm finds all vertices of the convex hull ordered along its boundary.

2.1 Algorithm

Let S = {P} be a finite set of points. The first step in this algorithm is to find the point with the lowest y-coordinate. If there is a tie, the point with the lowest x-coordinate out of the tie breaking candidates should be chosen. Call this point P_0. This step takes O(n), where n is the number of points in question.

Next, the set of points must be sorted in increasing order of the angle they and the point P_0 make with the x-axis. If there is a tie and two points have the same angle, discard the one that is closest to P_0. Any general-purpose sorting algorithm is appropriate for this, for example heapsort (which is O(n log n)). In order to speed up the calculations, it is not actually necessary to calculate the actual angle these points make with the x-axis; instead, it suffices to calculate the tangent of this angle, which can be done with simple arithmetic.

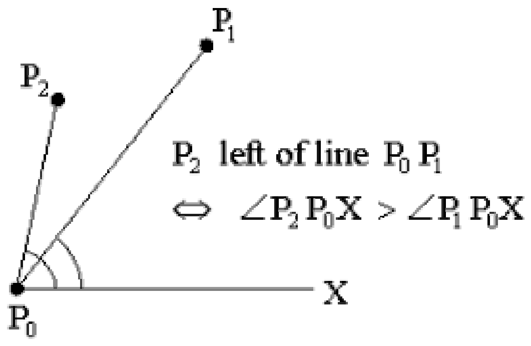
P_1
P_2
P_2 left of line $P_0 P_1$
$\Leftrightarrow \angle P_2 P_0 X > \angle P_1 P_0 X$
P_0
X

Fig. 1

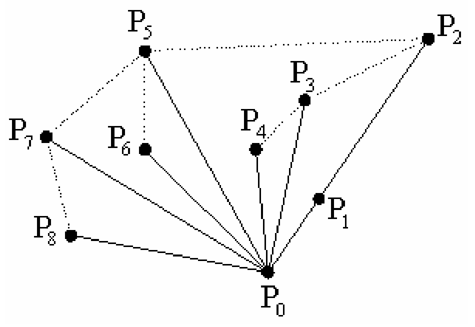

Fig. 2

The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is determined whether moving from the two previously considered points to this point is a "left turn" or a "right turn". If it is a "right turn", this means that the second-to-last point is not part of the convex hull and should be removed from consideration. This process is continued for as long as the set of the last three points is a "right turn". As soon as a "left turn" is encountered, the algorithm moves on to the next point in the sorted array. (If at any stage the three points are collinear, one may opt either to discard or to report it, since in some applications it is required to find all points on the boundary of the convex hull.)

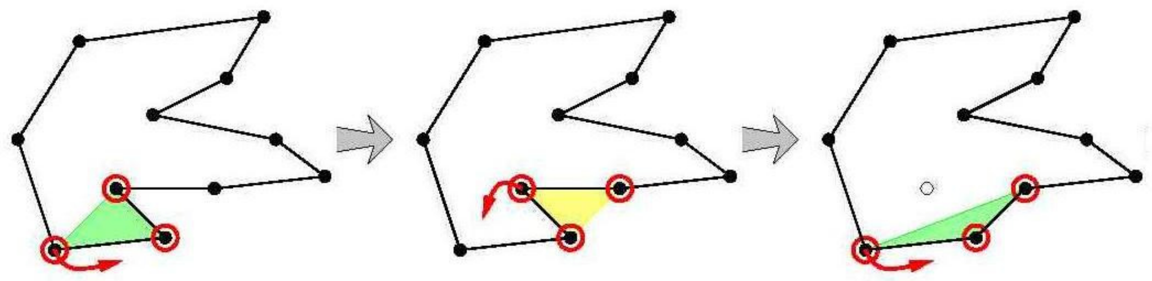

Fig. 3

Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only. For three points (x1,y1), (x2,y2) and (x3,y3), simply compute the direction of the cross product of the two vectors defined by points (x1,y1), (x2,y2) and (x1,y1), (x3,y3), characterized by the sign of the expression (x2 – x1)(y3 – y1) –

(y2 − y1)(x3 − x1). If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn", otherwise a "right turn".

This process will eventually return to the point at which it started, at which point the algorithm is completed and the array now contains the points on the convex hull in counterclockwise order.

2.2 Pseudocode

```
Find pivot P;
Sort Points by angle (with points with equal angle further sorted by distance from P);

# Points[1] is the pivot
Stack.push(Points[1]);
Stack.push(Points[2]);
FOR i = 3 TO Points.length
        WHILE Cross_product(Stack.second, Stack.top, Points[i])<=0 and Stack.length>=2
                Stack.pop;
        ENDWHILE
        Stack.push(Points[i]);
NEXT i

FUNCTION Cross_product(p1, p2, p3)
        RETURN (p2.x - p1.x)*(p3.y - p1.y) - (p3.x - p1.x)*(p2.y - p1.y);
ENDFUNCTION
```

The result will be stored on Stack.

2.3 Time complexity of the Graham's scan

Sorting the points has time complexity O(n log n). While it may seem that the time complexity of the loop is O(n^2), because for each point it goes back to check if any of the previous points make a "right turn", it is actually O(n), because each point is considered only once. Each point considered either terminates the inner loop, or it is removed from the array and thus never considered again. The overall time complexity is therefore O(n log n), since the time to sort dominates the time to actually compute the convex hull.

3 Jarvis's March

Graham's scan achieves O(nlogn) running time in the Euclidean plane E^2. However, if h, the number of hull vertices, is small, then it is possible to obtain better time bounds. A simple algorithm called Jarvis's march can construct the convex hull in O(nh) time.

3.1 Algorithm

The algorithm uses a technique called “gift wrapping,” which was first suggested by Chand and Kapur in 1970, after their research in computing convex hulls in arbitrary dimensions. This is a valuable algorithm when it is known that the convex hull will have few nodes in the perimeter set. For example, if the distribution of points was heavily clustered with a few extreme points that would comprise the outside perimeter. In real-world data, such distributions are not uncommon, and this method can capitalize on this attribute to compute the convex hull more rapidly. The algorithm’s concept is analogous to tying a string to a top most pin on a board, and wrapping the cord around the entire set of pins back to the first node. This is where the term “gift wrapping” originates. The shape of the cord would be the perimeter of the points in the set, or the convex hull.

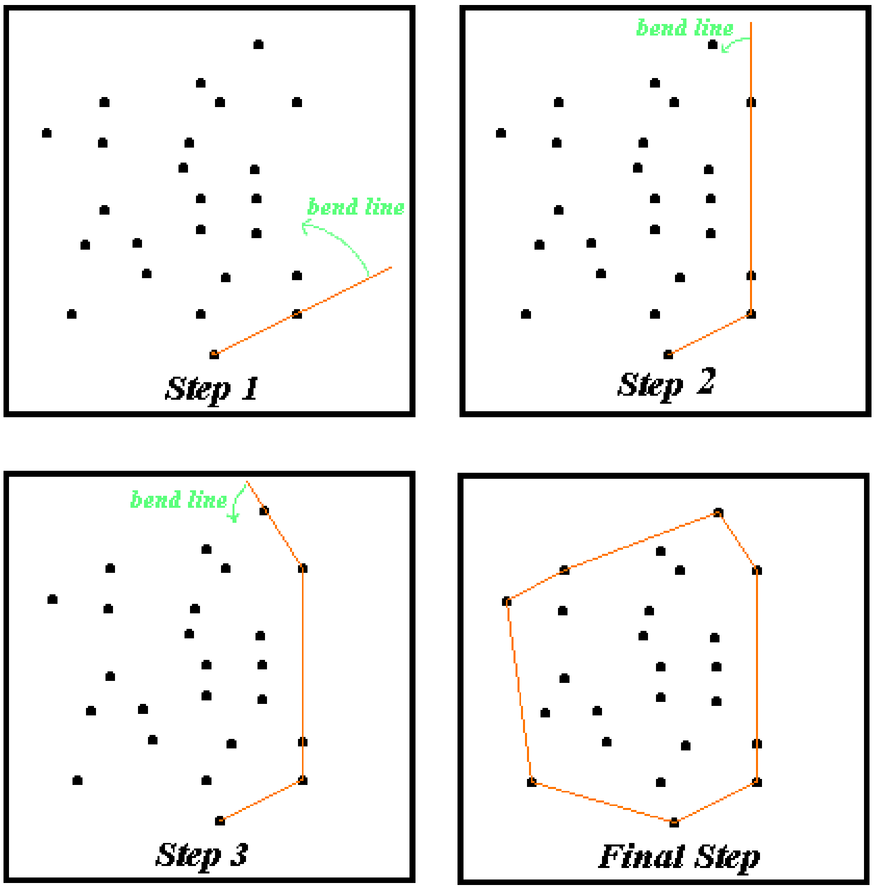

Fig. 4

Figure 4 provides a diagrammatic representation of the gift wrapping algorithm. The algorithm begins by locating the lowest (right-most) point, and then finds the point that has the smallest angle (from horizontal) from this point. A hull edge must join these two points (Figure 4, Step 1). The algorithm then proceeds to find the point with the smallest angle from this established hull edge. In effect, the line extending from the established hull edge is 'bent' around, in a counter-clockwise sense, until it touches another point. If two points are found to have the same angle, then the one furthest from the 'pivot' point is used (Figure 4, Step 2). The process of 'wrapping' the line around the set of points continues (Figure 4, Step 3) until the lower rightmost point is again reached (Figure 4, Final Step), at which point the convex hull has been discovered.

3.2 Pseudocode

```
Find the lowest (right-most) point
Let i(0) be its index, and set i=i(0)
repeat
    for each j =/= i do
        Compute ccw angle from previous hull edge
    Let k = index of the point with the smallest angle
    Output (p(i),p(k)) as a hull edge
    i = k
until i = i(0)
```

3.3 Time complexity of the Jarvis’s March algorithm

At each step, we need to consider all point as candidates for the convex hull. We can not exclude points that are classified at an early phase since these points might become convex hull points at a later phase. Therefore, the algorithm spends O(n) time for each convex hull vertex. Suppose there are h vertices on the convex hull, then the running time is O(nh). In the best case h = 3, the algorithm only needs O(n) time. In the worst case when all the points are on the convex hull, the time complexity of the algorithm is O(n^2). Jarvis’s March is “output-sensitive” algorithm, because the running time depends on number of hull vertices and not just on n; the smaller the output, the faster the algorithm.

4 Chan’s Algorithm

Chan’s algorithm is an optimal output-sensitive algorithm to compute the convex hull of n points in 2 or 3 dimensional space. The algorithm is never slower than either Jarvis’s march or Graham’s scan. The algorithm was discovered by Timothy Chan in 1993, and its running time is O(nlogh). Chan’s algorithm is a combination of divide-and-conquer and gift-wrapping.

4.1 Algorithm

The problem with Graham's scan is that it sorts all the points, and hence is doomed to having an O(n log n) running time, irrespective of the size of the hull. On the other hand, Jarvis's march can perform better if you have few vertices on the hull, but it takes O(n) time for each hull vertex.

Chan's idea was to partition the points into groups of equal size. There are m points in each group, and so the number of groups is r = ceil(n/m). For each group, compute its hull using Graham's scan, which takes O(m log m) time per group, for a total time of O(rm log m) = O(n log m). Next, run Jarvis's march on the groups. Here we take advantage of the fact that you can compute the tangent between a point and a convex m-gon in O(logm) time. So, as before there are h steps of Jarvis's march, but because we are applying it to r convex hulls, each step takes only O(r log m) time, for a total of O(hr log m) = ((hn/m) log m)time. Combining these two parts, we get a total of O((n + (hn/m)) log m) time. Observe that if we set m = h then the total running time will be O(nlog h), as desired.

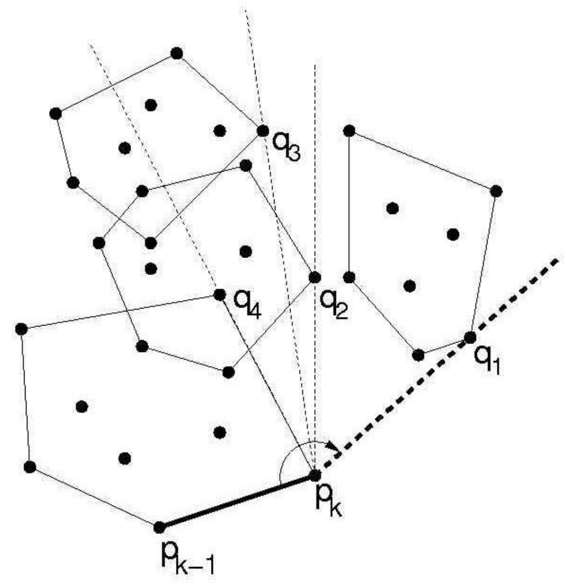

Fig. 5

4.2 Pseudocode

Chan's Partial Convex Hull Algorithm (Given m)

```
PartialHull(P; m) :
(1) Let r = ceil(n/m). Partition P into disjoint subsets P(1),P(2),... P(r), each of size at most m.
(2) For i = 1 to r do:
    (a) Compute Hull(P(i)) using Graham's scan and store the vertices in an ordered array.
(3) Let p0 = (-Inf; 0) and let p1 be the bottommost point of P .
(4) For k = 1 to m do:
    (a) For i = 1 to r do:
        Compute point q in P(i) that maximizes the angle p(k-1) p(k) q
    (b) Let p(k+1) be the point q in q(1),q(2),...q(r) than maximizes the angle p(k-1) p(k) q
    (c) If p(k+1) = p(1) then return {p(1), p(2), ... p(k)}.
(5) Return ``m was too small, try again."
```

We assume that we store the convex hulls from step (2a) in an ordered array so that the step inside the for loop of step (4a) can be solved in O(logm) time using binary search.

The only question remaining is how do we know what value to give to m? The last trick, is a common trick used in algorithms to guess the value of a parameter that affects the running time. We could try m = 1, 2, 3, ... , until we luck out and have m >= h, but this would take too long. Binary search would be a more efficient option, but if we guess to large a value for m (e.g. m = n/2) then we are immediately stuck with O(n log n) time, and this is too slow.

Instead, the trick is to start with a small value of m and increase it rapidly. Since the dependence on m is only in the log term, as long as our value of m is within a polynomial of h, that is, m = h^c for some constant c, then the running time will still be O(n log h). So, our approach will be to guess successively larger values of m, each time squaring the previous value, until the algorithm returns a successful result. This trick is often called doubling search (because the unknown parameter is successively doubled), but in this case we will be squaring rather than doubling.

Chan's Complete Convex Hull Algorithm

```
Hull(P ) :
(1) For t = 1; 2; : : : do:
(a) Let m = min(2^(2^t),n)
(b) Invoke PartialHull(P, m), returning the result in L.
(c) If L != ``try again" then return L.
```

4.3 Time complexity of the Chan's Algorithm

Note that 2^2^t has the effect of squaring the previous value of m. How long does this take? The t-th iteration takes O(n log 2^2^t) = O(n* 2^t) time. We know that it will stop as soon as 2^2^t >= h, that is if t = ceil(lg lg n). (We will use lg to denote logarithm base 2.) So, the total running time (ignoring the constant factors) is:

$$\sum_{t=1}^{\lg\lg h} n2^t = n\sum_{t=1}^{\lg\lg h} 2^t \leq n2^{1+\lg\lg h} = 2n\lg h = O(n\log h)$$

5 Dynamic Convex Hull Algorithm

It is easy to construct an example for which the convex hull contains all input points, but after the insertion of a single point the convex hull may become a triangle. And conversely, the deletion of a single point may produce the opposite drastic change of the size of the convex hull. Above mentioned algorithms are static and inefficient to maintain the convex hull in the dynamic situation. They need to recompute the convex hull each time the set changes. Overmars and Van Leeuwen introduce a dynamic algorithm in maintenance, i.e. keeping track, of the convex hull for the dynamic changing input data, with running time O($\log^2$n) per update.

5.1 Overmars and Van Leeuwen Algorithm

Overmars and Van Leeuwen Algorithm represents the convex hull as the union of two structures: the lc-hull and the rc-hull. The lc-hull is a convex arc that consists of the left faces of the convex hull. On the other hand, the rc-hull is a convex arc that consists of the right faces of the convex hull. Instead of updating the complex convex hull, Overmars and Van Leeuwen update the lc and rc-hulls by means of a divide-and-conquer framework. We focus at the lc-hull now.

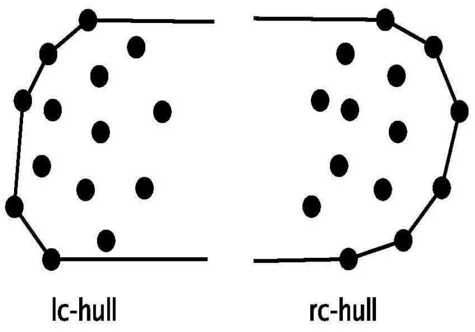

Fig. 6

In order to achieve the O(log^{2}n) running time, the following operations must be implemented efficiently (to take O(logN) time, where N is the number of points in the current convex hull) for lc-hull and rc-hull:
Search for a point
Split into two sub lc-hull
Join two lc-hulls
Insert a point.
The points of the lc-hull are stored in a concatenalbe queue sorted by their y-corrdinate.
A concatenable queue can perform these operations in just O(logN) time

Construction of the lc-hull

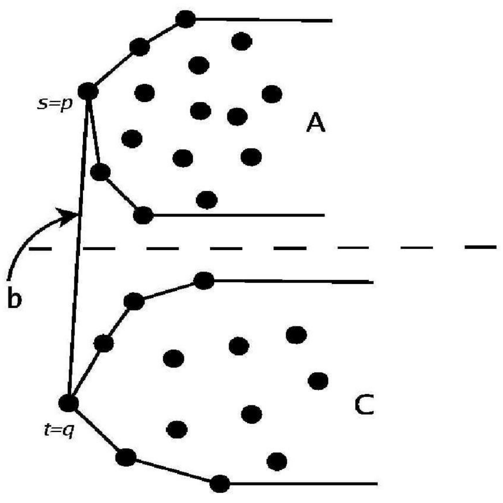

Fig. 7

The lc-hull is a structure that is decomposable in the following sense. Split the set of points P (with its points ordered by y-coordinate) by a horizontal line into two parts A and C, and in Fig. 7. the lc-hull of P is composed of portions of the lc-hulls of A and C, and a bridge B connecting the two parts.

Building the bridge

To build a bridge to connect two disjoint lc-hull, we need to find the tangent T to the lc-hull of A and C, and also find the points s, in the lc-hull of A, and t, in the lc-hull of C. The points s and t are the points where T intersects the lc-hull of A and C respectively. The task of finding the tangent T is done by doing a binary search along both of the hulls. Let p_1, p_2, ...,p_m be the points of the lc-hull of A ordered by their y-coordinate. Let q_1, q_2, …, q_k be the points of lc-hull of C order by their y-coordinate. We start with the points $p = p_{(m/2)}$ and $q = q_{(k/2)}$. We then check if the line that passes through p and q is tangent to the lc-hulls. If it is, we have found s and t. If the line is not a common tangent, Overmars and Van Leeuwen develop a criterion that enables us to eliminate the parts before or after p and q. Repeatedly choosing p and q in the middle of the remaining parts of A and C enables us to find p and q, hence bridge B, in O(logn). Let us look at the way pq intersects A and C. The following cases can occur.

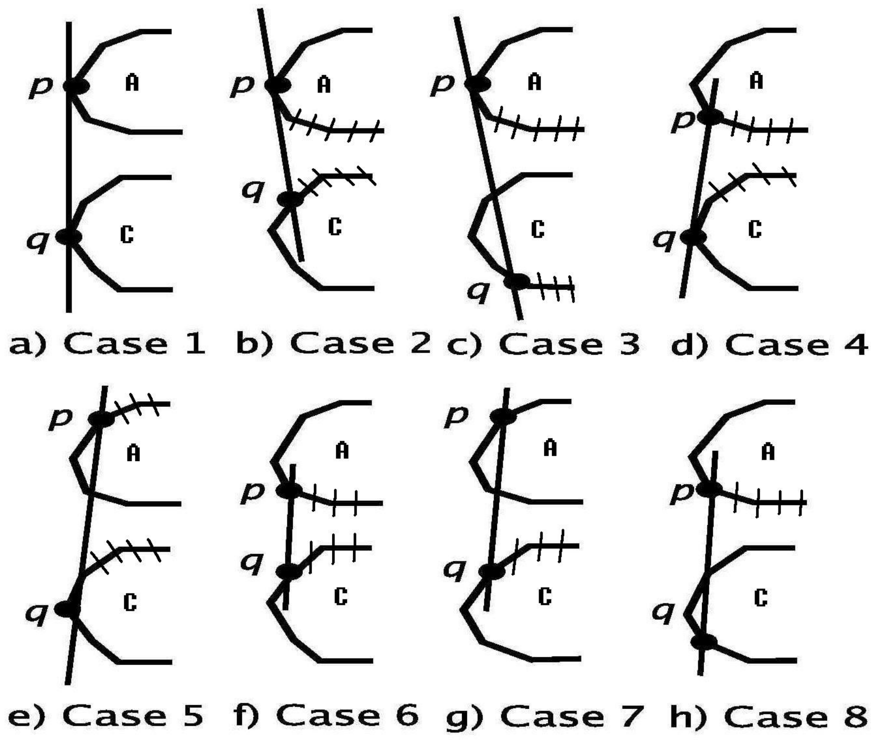

Fig. 8

Case 1, pq is the common tangent, we are done.
Case 2, the lower part of A after p, and the upper part of C before q can be deleted, search the remaining of A and C to find common tangent.
Case 3, the lower part of A after P, and the lower part of C after q can be deleted, search the remaining of A and C to find common tangent.

Case 4, the lower part of A after p, and the upper part of C before q can be deleted, search the remaining of A and C to find common tangent.
Case 5, the upper part of A before p, and the upper part of C before q can be deleted, search the remaining A and C to find common tangent.
Case 6, the lower part of A after p, and the upper part of C before q can be deleted, search the remaining of A and C to find common tangent.
Case 7, only the upper part of C before q can be deleted, search the remaining of A and C to find the common tangent.
Case 8, only the lower part of A after p can be deleted, search the remaining of A and C to find the common tangent.

Structuring the lc-hull

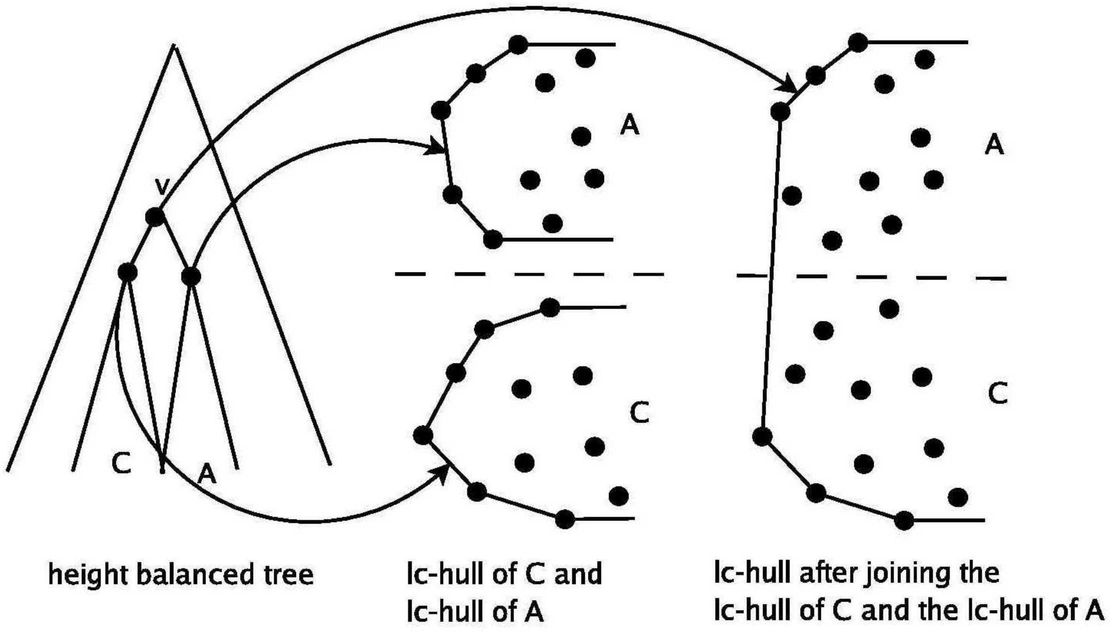

Fig. 9

The data structure that holds the lc-hulls and contains the information needed to deal with insertions and deletions is organized as follows. Its skeletal component is a height-balanced binary search tree whose leaves store the points of the current set, sorted by y-coordinate. The tree also stores in the internal nodes the lc-hull (and rc-hull) of the set of points in the subtree below that node. Keeping the points ordered by their y-coordinate allow Overmars and Van Leeuwen to recursively split the set in halves (the points stored in the left subtree of a node and the ones stored in the right subtree of a node) to build the lc-hull of the entire set. The height of tree is O(logn). The depth of all the leaves is O(logn).

The lc-hulls are organized as follows. A node v of the height-balanced binary search tree will have a concatenable queue that stores the lc-hull of the set formed by points stored in leaves in its subtree. A leaf will have the lc-hull of the set formed only by the point stored in it, which is a trivial lc-hull consisting of just that point.

The set of points P stored in the subtree of v is divided into two halves: the set of points stored in the left subtree, C, and the set of points stored in the right subtree, A. Since the points are ordered by y-coordinate, the points in the left subtree correspond to the points below some line parallel to the y axis. On the other hand, the points in the right subtree are above this line. We can build the lc-hull of P from the lc-hulls of C and A which are respectively stored in the left and right children of v.

Overmars and Van Leeuwen [5] do not retain the complete lc-hulls of the child nodes of v. Instead, they simply store the segment that is not part of the lc-hull stored in v. They do this to achieve the O($\log^2$ n) running time because it will take more than O(log n) time to copy the segments back to form the complete lc-hulls of the children. After splitting the lc-hulls of the children, the part of the lc-hull of C that is not used to form the lc-hull of P is stored in a concatenable queue in v's left child. Likewise, the part not used of the lc-hull of A is stored in v's right child. In node v, they also store the points of the bridge used to form the lc-hull. To fully reconstruct the lc-hulls of the children of v, they just split the lc-hull stored in v at the bridge and join each piece with the part stored in the corresponding children: the upper part of the lc-hull w.r.t. the bridge with the part stored in the right child and the lower part of the lc-hull w.r.t. the bridge with the part stored in the left child.

After the set and its lc-hull have been updated (after an insertion or deletion), the nodes in the balanced tree store the following. The root stores in a concatenable queue the complete lc-hull of the set and the points where the bridge connects the root's children lc-hulls. All the other nodes store the part of the lc-hull of the set of points stored at the leaves of its subtrees that does not form part of the lc-hull of set of points corresponding to its parent. If the lc-hull of the set of points stored in the leaves of the subtrees of a node is all part of the lc-hull of the set of points corresponding to the parent, then the concatenable queue stored in that node is an empty one. The nodes also store the points of the bridge that connect the lc-hulls of the set of points stored in the leaves of its left and right subtree.

Maintaining the Convex Hull

To insert or delete a point of the set of points and update the lc-hull of the set the process is the following. Overmars and Van Leeuwen [5] use two routines for it: DOWN and UP. They use the DOWN procedure to go down the tree in search of the leaf that stores the point to be deleted or the node where the point should be inserted. As they go down, at each node they explicitly build the lc-hulls of its two children. Once the point is inserted/deleted, they use the routine UP to go up the tree updating the lc-hulls all the way to the root, that has the complete lc-hull. To update the lc-hull at each node, they determine the bridge to join the lc-hull of the set of points stored in the leaves left subtree and the one of the set of points stored in the leaves of the right subtree.

The rc-hull is treated in the same way as the lc-hull. So during the DOWN and UP routines, the rc-hull is also updated. To update the convex hull of the current set, it is just the trivial join of the points that belong to the lc and rc-hulls after performing the DOWN and UP routines.

Running Time Analysis

Let the current set of points have n points and the current lc-hull N points. N is at most equal to n, so N = O(n). Since the top-level tree is a height-balanced tree, the procedure DOWN takes at most O(log n) steps to get to the leaf node that stores the point that is deleted or where the leaf node with new point must be inserted. At each step, we built the lc-hulls of the children of the current node v. To do that, as explained before, we must split the lc-hull stored in v at the bridge. Since the lc-hull is stored in a concatenable queue, that operation takes O(logN). Then we must join the parts resulting after splitting the lc-hull with the ones stored in the children. Each of those operations take O(logN). At each step of the DOWN routine, we perform work that costs O(logN) time. Since N = O(n), then the time spent at each step is O(log n). The DOWN routine finishes in $O(\log^2 n)$ time.

After using DOWN, they use UP to update the lc-hull. It will take O(log n) steps to reach to the root. At each step it updates the lc-hull of the node v it is visiting. To do this it first determines the bridge to join the two lc-hulls stored in v's children. This is done in O(logN) time, since it is a binary search over the two concatenable queues that store the lc-hulls of the children. Then it splits those concatenable queues taking O(logN) time. Finally, it joins the corresponding parts from the lc-hulls of the children and that operation takes O(logN). So each step of UP does work that takes O(logN). Like in DOWN, it is O(log n), and the total time that it takes to finish UP is $O(\log^2 n)$.

Since to update the rc-hull they do the same operations needed to update the lc-hull, both hulls are updated in $O(\log^2 n)$. To join the lc and rc-hulls it takes constant time.

If we use Overmars and Van Leeuwen's algorithm to compute the convex hull of set of n, it will be done on-line by first getting the convex hull of a set of one element. Then updates the convex hull of the set of one element after the insertion of the second point. Then it will update the convex hull of the set of two elements with the insertion of the third point, and so on. So when inserting the i+1 point, it will take O(log i) time to update the existing convex hull. After this update, we will have the convex hull of the first i + 1 elements. So the total time to compute the convex hull of the n points is:
$O(\log^2 2) + O(\log^2 3) + O(\log^2 4) + \dots + O(\log^2(n-1))$
From this we get that the total running time is: $O(n \log^2 n)$

So, to build the convex hull of a set of n points, we are paying a higher price than less powerful techniques, like Graham's Scan, that take O(n log n) time to compute the convex hull of a set of n points. But, if we are dealing with a dynamic set, it is clear that each update with the running time $O(\log^2 n)$ is faster than computing again the convex hull of the new set with another algorithm.

6 Comparing Running time

Randomly generate 100 points in the plane, we compare the running time of Graham's Scan with Overmars and Van Leeuwen's dynamic algorithm based on their time complexities.

Operation	Number of points in the set	Graham's Scan $O(n \log n)$	Overmars and Van Leeuwen $O(n\log^2 n)$ for initialization $O(\log^2 n)$ per update
Randomly generate 100 points in plane	100	664.38	4414.01
Insert 1 point	101	672.48	44.09
Delete 1 point	100	664.38	44.36

Graham's Scan is faster to compute the convex hull of the initial set of points. Overmars and Van Leeuwen's algorithm was faster to maintain the convex hull than re-computing it with Graham's Scan.

7 Conclusion

Graham's scan computes the convex hull of a given set of n points in the plane with time complexity O(n log n). However, if h, the number of hull vertices, is small, then it is possible to use Jarvis' March to construct the convex hull in O(nh) time. Chan provides an optimal output-sensitive algorithm, which is never slower than either Graham's scan and Jarvis's march. Chan's algorithm is a combination of divide-and-conquer and gift-wrapping, and its running time is O(n log h). These algorithms are static, and a modification to the points set requires to re-compute the convex hull from scratch.

Overmars and Van Leeuwen proposes a dynamic algorithm that can maintain the convex hull of a set after the insertion of a point into the set or the deletion of a point from the set, without re-computing the convex hull from scratch. It handles insertions and deletions in $O(\log^2 n)$ time, where n is the number of points in the set.

It might be worth attempting to construct an implementation that uses an algorithm that computes the convex hull of a set in O(n log n) time to compute the initial set of points and then handle the insertions and deletions with Overmars and Van Leeuwen's algorithm in $O(\log^2 n)$ time. This would allow the algorithm to compute the convex hull of the initial set more quickly, and also to be able to maintain it dynamically. This is a goal for my future work.

Bibliography

[1] T. M. Chan, Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions, *Discrete Computational Geometry*, 16:361-368, 1996.

[2] R. L. Graham. An efficient algorithm for determining the convex hull of a finite point set. *Inform. Process Letters*, 1:132-133, 1972.

[3] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process Letters*, 2:18-21, 1973.

[4] M. H. Overmars. *The Design of Dynamic Data Structures. Lecture Notes in Computer Science*, vol. 156. Springer-Verlag, 1983.

[5] M. H. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. System Science*, 23:166-204, 1981.