Comp 163: Computational Geometry — Professor Diane Souvaine
Tufts University, Spring 2005 — Scribe: Katelyn Mann

Dynamic Convex Hull and Order Decomposable Problems

1 Divide and Conquer Approach

In order to find the convex hull using a divide-and-conquer approach, follow these steps:

- sort points $(p_1, p_2, \ldots, p_n)$ by their x-coordinate
- recursively find the convex hull of p_1 through $p_{\frac{n}{2}}$
- recursively find the convex hull of $p_{\frac{n}{2}+1}$ through p_n
- merge the two convex hulls

The merge procedure requires finding a bridge between two given hulls which sit side by side. One way to do this is to find the bridge for the upper hull and the lower hull separately. This procedure takes advantage of the fact that the points are presorted by x-coordinate. Because of this ordering, it is easy to divide a group of points in half. See figure 1 for a view of how the merge procedure builds the bottom half of the convex hull. The top hull can be built similarly. The three main phases of the procdedure are:

- look for bridge points between the hulls using a varient of binary search
- split the hulls at the identified bridge points
- concatenate left part of left hull and right part of right hull

When searching for a bridge, any vertex could be the bridge point. When two vertices, a and b of hulls A and B, are being considered there are a number of different cases to look at to decide whether or not the $\overline{ab}$ is the correct bridge. The cases come from looking at the angles: $\angle(\overline{ab}, \vec{a_l}), \angle(\overline{ab}, \vec{a_r}), \angle(\vec{b_l}, \overline{ba}), \angle(\vec{b_r}, \overline{ba})$. These are the cases:

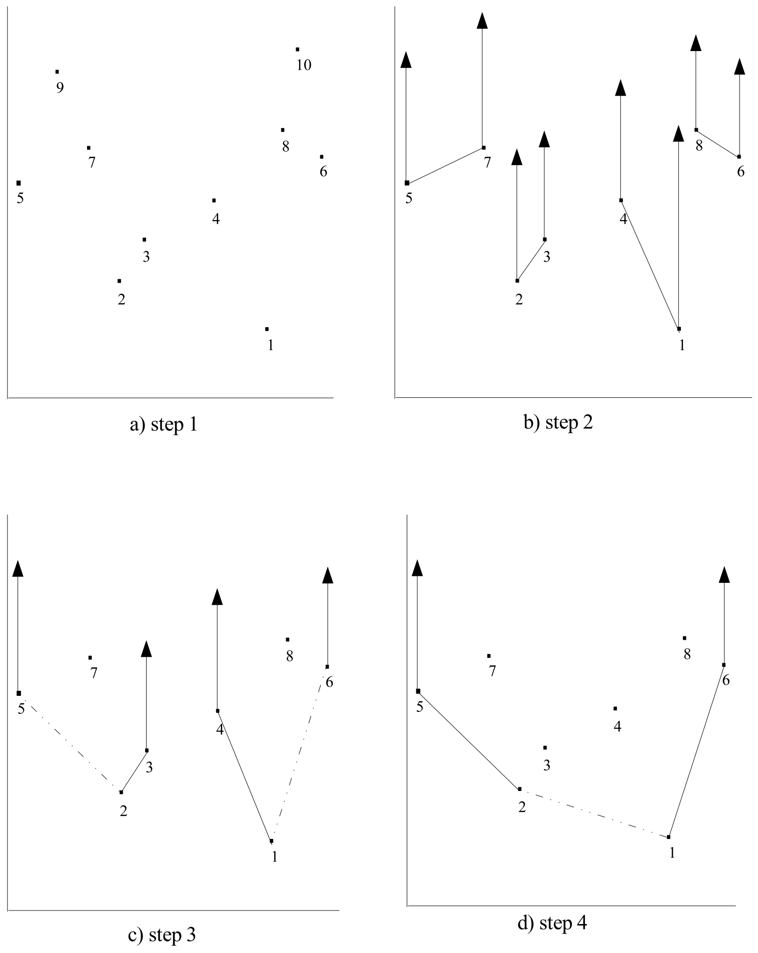

Figure 1: The merge procedure

1. If all of the four angles are $\leq$180 degrees, then $\overline{ab}$ is the bridge so we are done.

2. If $\angle(\overline{ab}, \vec{a_l}) \geq 180$ degrees, then neither a nor any vertex to the right of a can be the bridge point. The bridge of A and B is the bridge between B and the left sub-chain of A. So, we delete the right portion of A including a and continue to find the bridge of B and the left portion of A.

3. If $\angle(\vec{b_r}, \overline{ba}) \geq 180$ degrees, then neither b nor any vertex to the left of b can be the bridge point. Delete the left portion of B including b and continue to find the bridge between the A and the right portion of B.

4. If both $\angle(\overline{ab}, \vec{a_r})$ and $\angle(\vec{b_l}, \overline{ba}) \geq 180$ degrees, consider the intersection point, $v = (x, y)$, of $\vec{a_r}$ and $\vec{b_l}$. Let M_A be the maximum x-coordinate of A and m_B be the minimum x-coordinate of B.

    (a) If $x < m_b$, then no vertex of B could lie below $\vec{a_r}$ and therefore, the left part of A, including a can be removed.

    (b) If $x > M_A$, then no vertex of A could lie below $\vec{b_l}$ and therefore, the right part of B including b can be removed.

    (c) If $M_A < x < m_B$, then both of the above two conditions hold. Therefore, we can delete both the left part of A including a, and the right part of B, including b.

5. If only $\angle(\vec{b_l}, \overline{ba}) < 180$ degrees, all other angles are non-reflex, so neither b nor any point right of b can lie on the bridg, since $\overline{ba}$ lies within $\angle(\vec{b_r}, \vec{b_l})$.

6. If only $\angle(\overline{ab}, \vec{a_r})$ >180 degrees, all other angles are non-reflex, so $\overline{ab}$ lies within $\angle(\vec{a_l}, \vec{a_r})$, so neither a nor any point to the left of a can lie on the bridge.

See figure 2 for a look at all of the cases.

Analysis While trying to find the bridge, at least half of the vertices of one of the hulls is being thrown out during each step. So at least $\frac{1}{4}$ of the vertices get thrown out each time. Therefore, the process takes $O(\log n)$ time. Using a concatenable queue, also called a Q-structure, contatenating the left portion of the left hull and the right portion of the right hull can be

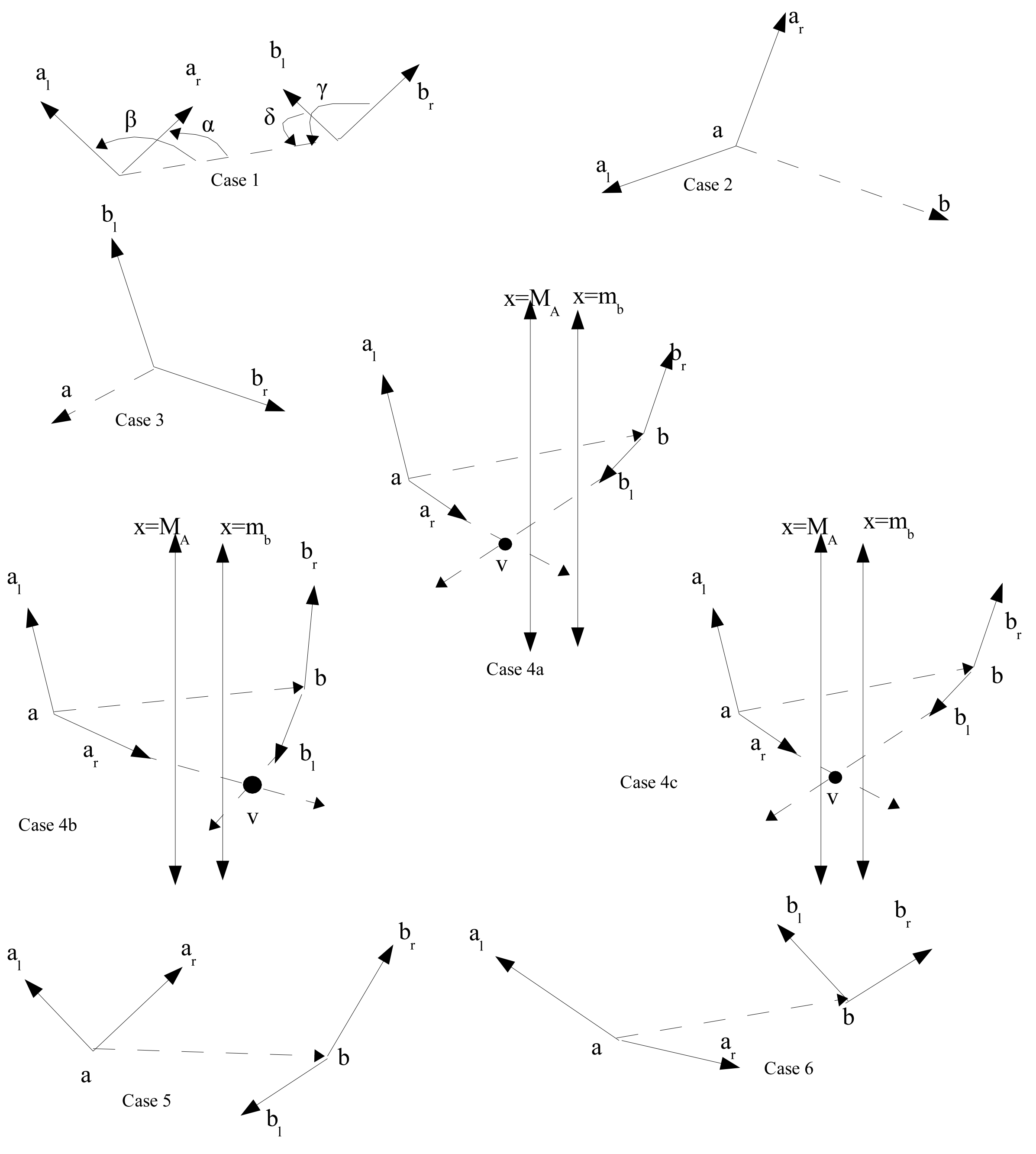

Figure 2: Cases for finding a bridge

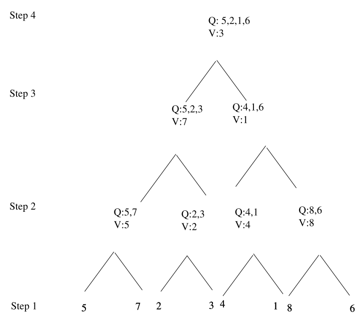

Figure 3: Divide-and-conquer tree

done efficiently. A concatenable queue is a form of a binary search tree that allows efficient searching, splitting and concatenating. Every internal node contains a pointer to the point with largest x-coordinate in its left subtree. The tree built for the previous example is shown in figure 3. So the merge procedure can be done in $O(\log n)$ using a concatenable queue and $O(n)$ otherwise. Since there are n vertices and $\log n$ levels in the tree the space complexity is $O(n \log n)$.

The first step of the convex hull algorithm was to sort all points by x-coordinate. That step takes $O(n \log n)$. Using what we know about the merge procedure, the recurrence is given by

$$T(n) = 2T(\frac{n}{2}) + O(\log n)$$

which is $O(n)$. Therefore, sorting the points dominates and the time complexity of the entire algorithm is $O(n \log n)$.

1.1 Review of Binary Trees

There are two models of binary trees. One in which the data sits in the leaf nodes and the other where the data sits in the internal nodes. See figure 4 for a look at the two different types. In the model containing data in the leaf nodes there can be pointers to the largest item in the left subtree as shown in figure 4. As mentioned above this pointer allows for more efficient performance of the concatenate, split, and search.

1.2 Supporting Lines

A *supporting line* is an extended line segment of a convex polygon that divides the plane such that the entire polygon lies to the same side of the line. The supporting line creates a *closed halfplane* on the side of the plane containing the polygon. See figure ̃reffig:sline.

2 Dynamic Convex Hull

The previous discussion of a divide-and-conquer approach to finding the convex hull assumed that all points were known ahead of time. In order to add or a remove a point from the set, the entire process would need to be repeated, taking $O(n \log n)$ time. The algorithm remains the same as before:

1. Sort the points by their x-coordinates. Let $v_1, v_2, \ldots, v_n$ denote the sorted list.
2. Form lower hull A of $v_1, v_2, \ldots, v_{\frac{n}{2}}$
3. Form lower hull A of $v_{\frac{n}{2}+1}, \ldots, v_n$
4. Merge A and B to form the lower hull of C

Almost the same merge procedure that was described in the previous section can be used here. The complexity for the initial build of the convex hull remains $O(n \log n)$. Since we are considering the hull to be dynamic, there needs to be an efficient way to insert and/or delete points. The tree used in the previous section holds information necessary to recover hulls before they were merged.

In the approach mentioned before the space complexity was $O(n \log n)$. Here a few changes can be made to achieve $O(n)$ space complexity. A vertex, v, could be a part of the convex hull at each level of the tree. Instead of wasting space and keeping track of that vertex at each level, it is only stored at the highest level that it reaches. So, at each node there is Q^*, B, and V. V remains the same as in the previous example - it is a pointer to the largest node in its left subtree. Q^* replaces Q at every level except for the top. Q^* is

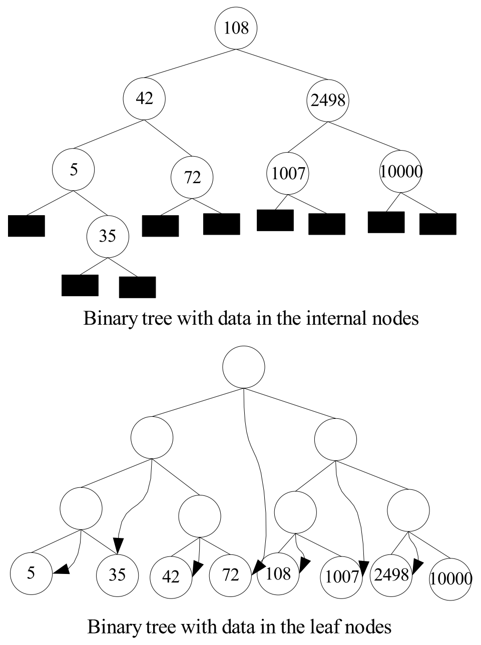
108
42
2498
5
72
1007
10000
35
Binary tree with data in the internal nodes
5
35
42
72
108
1007
2498
10000
Binary tree with data in the leaf nodes

Figure 4: 2 models of binary trees

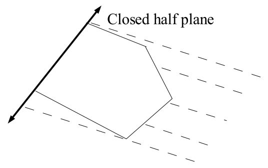

Figure 5: A supporting line

a pointer to a tree containing all points that are not part of the convex hull at the current level but were part of the hull in the previous level. B is the position of the bridge between the left convex hull and the right convex hull. See figure 6 for the dyamic version of the divide-and-conquer tree using the same example again. Note that in this small example, Q^* is not tree. In a more complex example, using more points, Q^* would take on a tree structure.

Given the tree structure, inserting a point into the set is the same as inserting a point as a leaf node of the tree then making the appropriate adjustments to the tree structure. In order to fix the tree, all convex hulls along the path from the inserted node to the root must be reconstructed. Starting at the top and using information in the tree to split along the path, the hulls are recovered. The process of inserting takes $O(\log^2 n)$. Splitting the tree takes $O(\log n)$ and the length of the path is $O(\log n)$ so the total time complexity of insertion is $O(\log^2 n)$.

3 Order Decomposable Problems

A problem is *M(n)-Order Decomposable* if it can be solved by the following steps:

- the set can be ordered by some scheme ORD

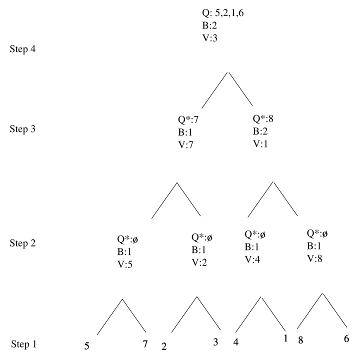

Figure 6: Dynamic Tree

- form a solution over the first i and last $n-i$ elements for all $1 < i < n$
- merge in time $M(n)$

A static divide-and-conquer algorithm takes time $O(ORD(n) + T(n))$ where $T(n) = 2T(\frac{n}{2}) + M(n)$. For example, the convex hull algorithm is $O(\log n)$-Decomposable and $T(n) = 2T(\frac{n}{2}) + O(\log n)$.

A dynamic operation that works on a balanced tree corresponds to the computation tree. The leaves correspond to the base case of the recursion and store the ordered elements of the set. Each nonleaf node α contains a pointer to the largest element in its left subtree, and stores the result of merging the results Q_β and Q_γ of its two children β and γ. As we merge Q_β and Q_γ, we also store the information for the merging in α. The unused pieces Q^*_β of Q_β and Q^*_γ and Q_γ are stored in β and γ.

To insert or delete an object, we locate the appropriate leaf using binary search. At each node α on the path from the root to the leaf, we use the merging information stored at α to split Q_α and use them together with Q^*_β and Q^*_γ to recover Q_β and Q_γ. The process of descending the tree takes $Down(n) = O(M(n))$ if $M(n) = \Omega(n^\epsilon)$ for $\epsilon > 0$, else $Down(n) = O(M(n) \log n)$.

After updating the leaf of the tree, we ascend the tree to reform the result along the path from the leaf to the root. Since the number of merging and splitting is the same, $Up(n) = Down(n)$.

From the above analysis, we conclude that the dynamic maintenace of the set can be performed in $O(M(n))$ if $M(n) = \Omega(n^\epsilon)$ for $\epsilon > 0$, otherwise it takes $O(M(n \log n)$.

3.1 Maximal Problem

Another example of an M(n)-Order Decomposable problem is the maximal problem.

Finding the maximal elements Max of a set of points in R^2 where elements in Max are sorted by the y-coordinate.

A point p is maximal in a sset of points, S if and only if either $x_p > x_q$ or $y_p > y_q$ for all $q \in S$. The problem can be solved by

- Sort S by x-coordinate. Let $p_1, p_2, \ldots, p_n$ donote a sorted list
- Find the maximal elements Max_l for $p_1, p_2, \ldots, p_{\frac{n}{2}}$
- Find the maximal elements Max_r for $p_{\frac{n}{2}+1}, \ldots, p_n$
- Merge Max_l and Max_r to form Max

The merge can be done in the following way: Let p_1 and p_r be the highest points of Max_l and Max_r respectively. if $p_l > p_r$, binary search for p_r in Max_l by the y-coordinate and split Max_l. Max is the higher portion of Max_l merged with Max_r Otherwise ($p_r \geq p_l$), $Max = Max_r$. The merge process can be done in $O(n \log n)$ by splitting and concatenation. So, this problem is $O(\log n)$-decomposable, and the dynamic maintenance of maximal elements of a set S could be done in $O(\log^2 n)$ time.

3.2 Finding the Lower Envelope

Find the intersection of n lower halfplanes (also called the lower envelope). Let the set of lower halfplanes be $H_i = \{(x, y) | y < a_i x + b_i\}$. The problem can be solved by the following steps:

1. Sort H_i by the slope of the boundary lines. Let $l_1, l_2, \ldots, l_n$ denote the sorted list.
2. Find the intersection of the lower halfplanes whose boundary lines are $l_1, l_2, \ldots, l_i$.

3. Find the intersection of the lower halfplanes whose boundary lines are $l_{i+1}, \ldots, l_n$.

4. Merge the results of the above.

The merge can be done in the following manner: Find the intersection of the two contours in time $O(\log n)$, split them there and concatenate the front piece of the first with the tail piece of the second, also using $O(\log n)$ time. Consequently, the lower halfplane problem is also $O(\log n)$-decomposable.