CMSC 754: Lecture 3
Convex Hulls: Lower Bounds and Output Sensitivity

Reading: Chan's output sensitive algorithm can be found in T. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions", *Discrete and Computational Geometry*, 16, 1996, 361–368.

Lower Bound and Output Sensitivity: Last time we presented two planar convex hull algorithms, Graham's scan and the divide-and-conquer algorithm, both of which run in $O(n \log n)$ time. A natural question to consider is whether we can do better.

Recall that the output of the convex hull problem a convex polygon, that is, a cyclic enumeration of the vertices along its boundary. Thus, it would seem that in order to compute the convex hull, we would "need" to sort the vertices of the hull. It is well known that it is not generally possible to sort a set of n numbers faster than $\Omega(n \log n)$ time, assuming a model of computation based on binary comparisons. (There are faster algorithms for sorting small integers, but these are not generally applicable for geometric inputs.)

Can we turn this intuition into a formal lower bound? We will show that in $O(n)$ time it is possible to reduce the sorting problem to the convex hull problem. This implies that any $O(f(n))$-time algorithm for the convex hull problem implies an $O(n + f(n))$-time algorithm for sorting. Clearly, $f(n)$ cannot be smaller than $\Omega(n \log n)$ for otherwise we would obtain an immediate contradiction to the lower bound on sorting.

The reduction works by projecting the points onto a convex curve. In particular, let $X = \{x_1, \ldots, x_n\}$ be the n values that we wish to sort. We will map this into a 2-dimensional point set by projecting the points onto the boundary of a convex shape, so that the sorted order is preserved. For example, suppose that we project each point vertically onto the parabola $y = x^2$, by mapping x_i to the point $p_i = (x_i, x_i^2)$ (see Fig. 1(a)). Let P denote the resulting set of points. It is easy to see that all the points of P lie on its convex hull, and the sorted order of points along the lower hull is the same as the sorted order X (see Fig. 1(b)). Once we obtain the convex hull as a cycle sequence of vertices, in $O(n)$ additional time we can extract its lower hull from left to right, thus obtaining X in sorted order (see Fig. 1(c)).

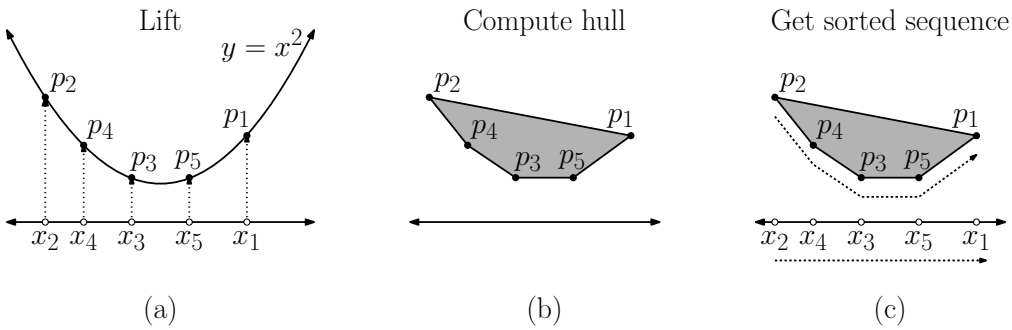

Fig. 1: Reduction from sorting to convex hull.

Theorem: Assuming computations based on comparisons (e.g., orientation tests) any algorithm for the convex hull problem requires $\Omega(n \log n)$ time in the worst case.

Is this the end of the story? Well, maybe not ...

- What if we don't require that the points be enumerated in cyclic order? For example, suppose we just wanted to count number of points on the convex hull. Can we do better?
- Suppose that we are not interested in worst-case behavior. For example, in many instances of convex hull, relatively few points lie on the boundary of the hull.

We will present three other results later in this lecture:

- We will present a convex hull algorithm, called Jarvis's March, that runs $O(nh)$ time, where h is the number of vertices on the hull. (This is beats Graham's algorithm whenever h is asymptotically smaller than $O(\log n)$.)
- We will present Chan's algorithm, which computes the convex hull in $O(n \log h)$ time.
- We will present a lower bound argument that shows that, assuming a comparison-based algorithm, even answering the question "does the convex hull have h distinct vertices?" requires $\Omega(n \log h)$ time.

The last result implies that Chan's algorithm is essentially the best possible as a function of h and n. An algorithm whose running time depends on the output size is called *output sensitive*. Both Jarvis's March and Chan's algorithm are output sensitive.

Jarvis's March: Our next convex hull algorithm, called *Jarvis's march*, computes the convex hull in $O(nh)$ time by a process called "gift-wrapping." In the worst case, $h = n$, so this is inferior to Graham's algorithm for large h, it is superior if h is asymptotically smaller than $\log n$, that is, $h = o(\log n)$.

Jarvis's algorithm begins by identifying any one point of P that is guaranteed to be on the hull, say, the point with the smallest y-coordinate. (As usual, we assume general position, so this point is unique.) Call this v_1. It then repeatedly finds the next vertex on the hull in counterclockwise order.

Given a triple of distinct points $\langle p, q, r \rangle$, define the *turning angle* of r with respect to p and q to be the (CCW) angle between the directed line $\overline{pq}$ and the directed line $\overline{qr}$ (see Fig. 2(a)).

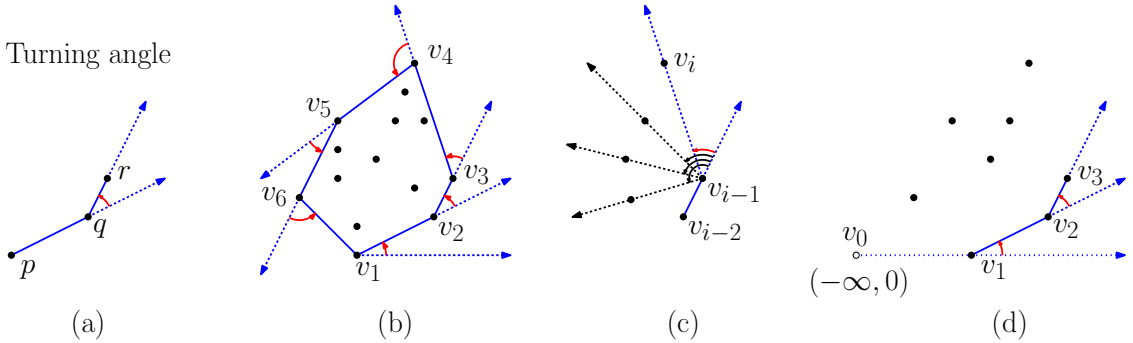

Fig. 2: Jarvis's march.

Jarvis's march works by repeatedly computing the next hull vertex v_i as the point of P that minimizes the turning angle with respect to the prior two, v_{i-2} and v_{i-1} (see Fig. 2(c)). Since

we need two points, to get the ball rolling, it is convenient to define an imaginary "sentinel point" $v_0 = (-\infty, 0)$, which has the effect that the initial line $\overline{v_0 v_1}$ is directed horizontally to the right (see Fig. 2(d)).

Jarvis's March

(1) Given P, let $v_0 = (-\infty, 0)$ and let v_1 be the point of P with the smallest y-coordinate

(2) For $i \leftarrow 2, 3, \ldots$

  (a) $v_i \leftarrow$ the point of $P \setminus \{v_{i-1}, v_{i-2}\}$ that minimizes the turning angle with respect to v_{i-2} and v_{i-1}

  (b) If $v_i == v_1$, return $\langle v_1, \ldots, v_{i-1} \rangle$

The algorithm's correctness follows from the fact that (by induction) $\overline{v_{i-2} v_{i-1}}$ is a CCW-directed edge of the hull, and hence the next vertex of the hull is the one that minimizes the turning angle.

By basic trigonometry, turning angles can be computed in constant time. But it is interesting to note that it is possible to compare turning angles just using orientation tests. (Try this yourself.) This implies that if the input coordinates are integers, the vertices of the hull can be computed exactly (assuming double-precision integer computations).

To obtain the running time, observe that v_1 can be computed in $O(n)$ time, and each iteration can be implemented in $O(n)$ time. After h iterations, the algorithm terminates, so the total running time is $O(n + nh) = O(nh)$.

Chan's Algorithm: Depending on the value of h, Graham's scan may be faster or slower than Jarvis' march. This raises the intriguing question of whether there is an algorithm that *always* does as well or better than these algorithms. Next, we present a planar convex hull algorithm by Timothy Chan whose running time is $O(n \log h)$.

While this algorithm is too small an improvement over Graham's algorithm to be of significant practical value, it is quite interesting nonetheless from the perspective of the techniques that it uses:

- It combines two slower algorithms, Graham's and Jarvis's, to form a faster algorithm.
- It employs a clever guessing strategy to determine the value of a key unknown parameter, the number h of vertices on the hull.

To gain some intuition behind Chan's algorithm, let us first observe that in order to replace the $O(\log n)$ factor in Graham's algorithm with $O(\log h)$, we cannot afford to sort any set whose size is (significantly) larger than h. This would seem impossible at first glance, since we don't know the value of h until the algorithm terminates! We will get around this by playing a "guessing game" for the value of h. We'll start low, and work up to successively guesses for what h is. Throughout, let h denote the true number of vertices on the hull. The algorithm will maintain a variable h^*, which is our current "guess" on the value of h. As we shall see, if we guess wrong, we will discover our error, and we will need to increase our guess. For now, let us assume that a magical little bird has told us the value of h^* that works.

We will need to make use of a *utility function*, whose implementation we will leave as an exercise. Recall that a *support line* for a convex body is a line that contacts the boundary

of the body and the body lies entirely on one side of the line. Given a convex body Q and any point p external to Q, there are exactly two support lines of Q that pass through p. The next lemma shows that we can compute them logarithmic time.

Lemma: Given a convex polygon $Q = \langle q_1, \ldots, q_m \rangle$, where the vertices are stored in an m-element array sorted in CCW order around Q's boundary, and given any point p that is external to Q, in $O(\log m)$ time we can compute the two vertices q^- and q^+ of Q so that $\overline{pq^-}$ and $\overline{pq^+}$ are support lines of Q.

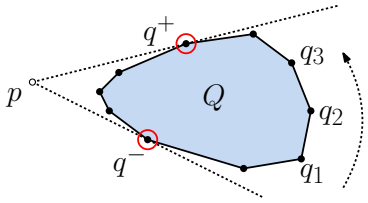

Fig. 3: Utility function - Computing tangent points.

We can now describe Chan's algorithm, conditioned on the fact that we have a guess h^* on the size of the hull.

Step 1: (Mini-hulls) Partition P (arbitrarily) into $k = \lceil n/h^* \rceil$ groups, each of size at most h^*. Call these $P_1, \ldots, P_k$ (see Fig. 4(b)). By Graham's algorithm, compute the convex hull of each subset. Let $H_1, \ldots, H_k$ denote the resulting *mini-hulls* (see Fig. 4(c)).

How long does this take? We can compute each mini-hull in time $O(h^* \log h^*)$. Applying this to each of the k groups, we have an overall time of $O(k(h^* \log h^*)) = O(n \log h^*)$. Note that if we guess the value of h correctly (that is, $h^* = h$) then this runs in time $O(n \log h)$, as desired.

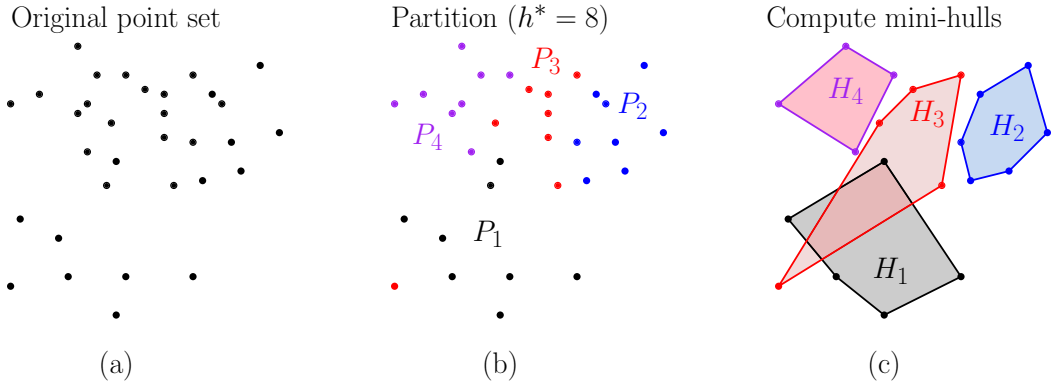

Fig. 4: Step 1: Partitioning and mini-hulls (for $k = 4$).

Step 2: (Merging) The high-level idea is to run Jarvis's march on the mini-hulls (see Fig. 5(a)). We will treat each mini-hull as if it is a "fat point". In particular, we take the most recent vertex v_{i-1} and for each mini-hull H_j employ the utility function of the above lemma to compute the vertices q_j^- and q_j^+ for the support lines for this

mini-hull (see Fig. 5(b)). As in Jarvis's algorithm, among all of these support points, we take v_i to be the one that minimizes the turn angle with respect to v_{i-2} and v_{i-1} (see Fig. 5(c)). (By the nature of Jarvis's algorithm, we only need to compute the support line with the smaller turning angle, but computing both will not affect the asymptotic running time. Also note that we do this for the mini-hull containing v_{i-1} itself, but this is trivial.)

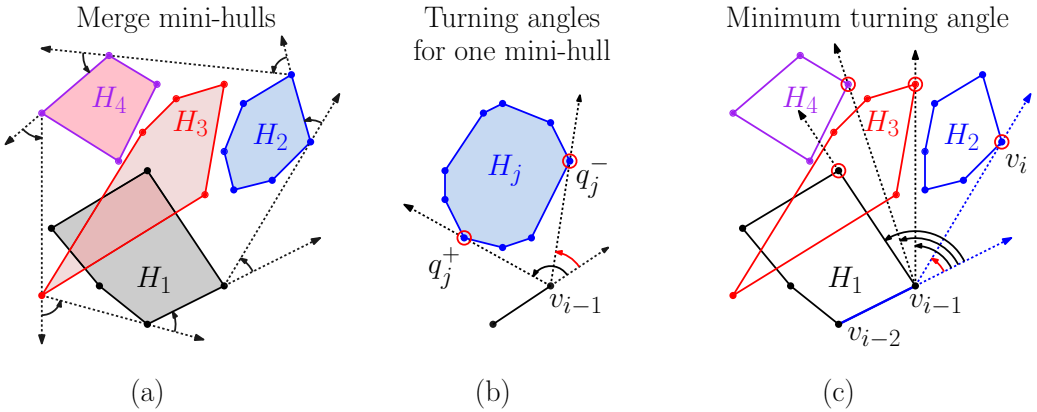

Fig. 5: Step 2: Merging the mini-hulls.

How long does this take? For each mini-hull, we can compute the two support lines in time $O(\log h^*)$ by the lemma. The number of support lines is twice the number of mini-hulls, so for each step of Jarvis's algorithm, we can compute all the relevant turning angles in time $O(k \log h^*)$. Each iteration of Jarvis yields one more vertex of the final convex hull (of which there are h), so the overall running time is $O(h(k \log h^*))$. Observe that if we managed to guess the right value of h (that is, if $h^* = h$), then this takes time $O(h^*(k \log h^*)) = O(n \log h^*) = O(n \log h)$, as desired.

In summary, we have argued above that, if we were lucky enough to guess the correct hull size ($h^* = h$), then the method outlined above will yield the convex hull in time $O(n \log h)$.

The Conditional Algorithm: We can now present a conditional algorithm for computing the convex hull. The algorithm is given a point set P and an estimate h^* of the number of vertices on P's convex hull. Letting h denote the actual number of vertices on the hull. We will see below that if the h^* is significantly larger or smaller than h, the algorithm will not be efficient. In the merge phase, if we ever see more than h^* hull vertices, we know that our estimate is too low, and we terminate the algorithm (returning "failure") before the damage is too great.

Guessing the Hull's Size: The conditional algorithm assumes that we have a good estimate in h^* for h. What are the consequences of guessing wrong?

Too large? If we guess a value of $h^* > h$, then the Graham scans, which together run in $O(n \log h^*)$ time may be too slow. Notice, however, that we pay only a constant factor even if h^* is polynomially larger than h. For example, if $h < h^* \leq h^2$, then the running time of Graham's scan is $O(n \log(h^2)) = O(2n \log h) = O(n \log h)$, which is okay for us.

Chan's Algorithm for the Conditional Hull Problem

ConditionalHull(P, h^*) :

(1) Let $k \leftarrow \lceil n/h^* \rceil$. Partition P (arbitrarily) into disjoint subsets $P_1, \ldots, P_k$, each of size at most h^*

(3) For $j \leftarrow 1$ to k, compute $H_j = \text{conv}(P_j)$ using Graham's scan, storing each in an ordered array

(4) Let $v_0 \leftarrow (-\infty, 0)$, and let v_1 be the bottommost point of P

(5) For $i \leftarrow 1, 2, \ldots, h^*$:

  (a) For $j \leftarrow 1$ to k, using the utility lemma, compute the tangents points q_j^- and q_j^+ for H_j with respect to v_{i-1}.

  (b) Set v_i to be the tangent point that minimizes the turning angle with respect to v_{i-2} and v_{i-1}

  (c) If $v_i = v_1$ then return the pair $(\textbf{success}, V = \langle v_1, \ldots, v_{i-1} \rangle)$

(6) If we get here, we know that $h^* < h$, and we return $(\textbf{failure}, \emptyset)$

Too small? If we guess a value of $h^* < h$, then the merge phase will be too slow. It is easy to verify that (even if we didn't stop it because of the failure condition) it would run in time $O(n(h/h^*) \log h^*)$. If our estimate h^* is not within a constant factor of h, then we will not achieve our desired running time. This is why we chose to use the "failure" option. Since we never do more than h^* iterations, the running time of each failure phase is just $O(n \log h^*)$.

Here is what we'll do. We'll start with a low estimate for h^* (e.g., $h^* = 3$). If the algorithm returns "failure", we increase h^* until we succeed. The question is how quickly we should step up the value of h^*?

It is easy to show that increasing h^* in an arithmetic progression (e.g., $h^* = 3, 4, 5, \ldots$) will be way to slow. A smarter approach is to grow h^* through doubling (e.g. $h^* = 4, 8, 16, \ldots, 2^i$). We will leave it as an exercise to show that this is also too slow. (It will lead to a running time of $O(n \log^2 h)$.)

Recall that we are allowed to overshoot the actual value of h by any polynomial. Let's try *repeatedly squaring* the previous guess. In other words, let's try $h^* = 2, 4, 16, 256, \ldots, 2^{2^i}$. Clearly, as soon as we reach a value for which the restricted algorithm succeeds, we have $h < h^* \leq h^2$. Therefore, the running time for this last stage will be $O(n \log h)$, as desired. But what about the total time for all the previous stages?

To analyze the total time, consider the ith guess, $h_i^* = 2^{2^i}$. The ith trial takes time $O(n \log h_i^*) = O\big(n \log 2^{2^i}\big) = O(n2^i)$. We know that we will succeed as soon as $h_i^* \geq h$, that is if $i = \lceil \lg \lg h \rceil$. (Throughout the semester, we will use "lg" to denote logarithm base 2 and "log" when the base does not matter.[1]) Thus, the algorithm's total running time (up to constant factors) is

$$T(n, h) \;=\; \sum_{i=1}^{\lg \lg h} n2^i \;=\; n \sum_{i=1}^{\lg \lg h} 2^i.$$

The summation is a geometric series. It is well known that a geometric series is asymptotically

[1] When $\log n$ appears as a factor within asymptotic big-O notation, the base of the logarithm does not matter provided it is a constant. This is because $\log_a n = \log_b n / \log_b a$. Thus, changing the base only alters the constant factor.

dominated by its largest term. Thus, we obtain a total running time of

$$T(n,h) \;<\; n \cdot 2^{\lceil \lg \lg h \rceil} \;<\; n \cdot 2^{1+\lg \lg h} \;=\; n \cdot 2 \cdot 2^{\lg \lg h} \;=\; 2n \lg h \;=\; O(n \log h),$$

which is just what we want. In other words, by the "miracle" of the geometric series, the total time to try all the previous failed guesses is asymptotically the same as the time for the final successful guess. The final algorithm is presented in the code block below.

Chan's Complete Convex Hull Algorithm

Hull(P) :

(1) $h^* \leftarrow 2$; status $\leftarrow$ failure
(2) while (status $\neq$ failure):
  (a) Let $h^* \leftarrow \min((h^*)^2, n)$
  (b) (status, V) $\leftarrow$ ConditionalHull(P, h^*)
(3) return V

Lower Bound (Optional): We show that Chan's result is asymptotically optimal in the sense that any algorithm for computing the convex hull of n points with h points on the hull requires $\Omega(n \log h)$ time. The proof is a generalization of the proof that sorting a set of n numbers requires $\Omega(n \log n)$ comparisons.

If you recall the proof that sorting takes at least $\Omega(n \log n)$ comparisons, it is based on the idea that any sorting algorithm can be described in terms of a *decision tree*. Each comparison has at most three outcomes ($<$, $=$, or $>$). Each such comparison corresponds to an internal node in the tree. The execution of an algorithm can be viewed as a traversal along a path in the resulting ternary (3-way splitting) tree. The height of the tree is a lower bound on the worst-case running time of the algorithm. There are at least $n!$ different possible inputs, each of which must be reordered differently, and so you have a ternary tree with at least $n!$ leaves. Any such tree must have $\Omega(\log_3(n!))$ height. Using Stirling's approximation for $n!$, this solves to $\Omega(n \log n)$ height. (For further details, see the algorithms book by Cormen, Leiserson, Rivest, and Stein.)

We will give an $\Omega(n \log h)$ lower bound for the convex hull problem. In fact, we will give an $\Omega(n \log h)$ lower bound on the following simpler decision problem, whose output is either yes or no.

Convex Hull Size Verification Problem (CHSV): Given a point set P and integer h, does the convex hull of P have h distinct vertices?

Clearly if this takes $\Omega(n \log h)$ time, then computing the hull must take at least as long. As with sorting, we will assume that the computation is described in the form of a decision tree. The sorts of decisions that a typical convex hull algorithm will make will likely involve orientation primitives. Let's be even more general, by assuming that the algorithm is allowed to compute *any* algebraic function of the input coordinates. (This will certainly be powerful enough to include all the convex hull algorithms we have discussed.) The result is called an *algebraic decision tree*.

The input to the CHSV problem is a sequence of $2n = N$ real numbers. We can think of these numbers as forming a vector in real N-dimensional space, that is, $(z_1, z_2, \ldots, z_N) = \vec{z} \in \mathbb{R}^N$, which we will call a *configuration*. Each node branches based on the sign of some function of the input coordinates. For example, we could implement the conditional $z_i < z_j$ by checking whether the function $(z_j - z_i)$ is positive. More relevant to convex hull computations, we can express an orientation test as the sign of the determinant of a matrix whose entries are the six coordinates of the three points involved. The determinant of a matrix can be expressed as a polynomial function of the matrices entries. Such a function is called *algebraic*. We assume that each node of the decision tree branch three ways, depending on the sign of a given multivariate algebraic formula of degree at most d, where d is any fixed constant. For example, we could express the orientation test involving points $p_1 = (z_1, z_2)$, $p_2 = (z_3, z_4)$, and $p_3 = (z_5, z_6)$ as an algebraic function of degree two as follows:

$$\det \begin{pmatrix} 1 & z_1 & z_2 \\ 1 & z_3 & z_4 \\ 1 & z_5 & z_6 \end{pmatrix} = (z_3 z_6 - z_5 z_4) - (z_1 z_6 - z_5 z_2) + (z_1 z_4 - z_3 z_2).$$

For each input vector $\vec{z}$ to the CHSV problem, the answer is either "yes" or "no". The set of all "yes" points is just a subset of points $Y \subset \mathbb{R}^N$, that is a region in this space. Given an arbitrary input $\vec{z}$ the purpose of the decision tree is to tell us whether this point is in Y or not. This is done by walking down the tree, evaluating the functions on $\vec{z}$ and following the appropriate branches until arriving at a leaf, which is either labeled "yes" (meaning $\vec{z} \in Y$) or "no". An abstract example (not for the convex hull problem) of a region of configuration space and a possible algebraic decision tree (of degree 1) is shown in the following figure. (We have simplified it by making it a binary tree.) In this case the input is just a pair of real numbers.

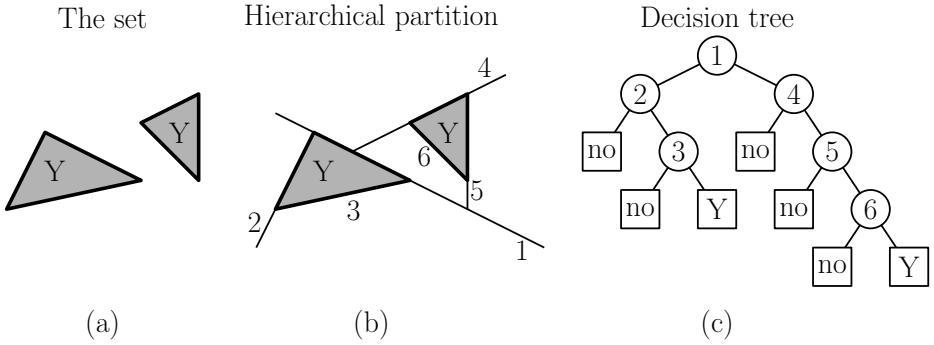

Fig. 6: The geometric interpretation of an algebraic decision tree.

We say that two points $\vec{u}, \vec{v} \in Y$ are in the same *connected component* of Y if there is a path in $\mathbb{R}^N$ from $\vec{u}$ to $\vec{v}$ such that all the points along the path are in the set Y. (There are two connected components in the figure.) We will make use of the following fundamental result on algebraic decision trees, due to Ben-Or. Intuitively, it states that if your set has M connected components, then there must be at least M leaves in any decision tree for the set, and the tree must have height at least the logarithm of the number of leaves.

Theorem: Let $Y \in \mathbb{R}^N$ be any set and let T be any d-th order algebraic decision tree that determines membership in W. If W has M disjoint connected components, then T must have height at least $\Omega((\log M) - N)$.

We will begin our proof with a simpler problem.

Multiset Size Verification Problem (MSV): Given a multiset of n real numbers and an integer k, confirm that the multiset has exactly k distinct elements.

Lemma: The MSV problem requires $\Omega(n \log k)$ steps in the worst case in the d-th order algebraic decision tree

Proof: In terms of points in $\mathbb{R}^n$, the set of points for which the answer is "yes" is

$$Y = \{(z_1, z_2, \ldots, z_n) \in \mathbb{R}^n \ : \ |\{z_1, z_2, \ldots, z_n\}| = k\}.$$

It suffices to show that there are at least $k!k^{n-k}$ different connected components in this set, because by Ben-Or's result it would follow that the time to test membership in Y would be

$$\Omega(\log(k!k^{n-k}) - n) \ = \ \Omega(k \log k + (n-k) \log k - n) \ = \ \Omega(n \log k).$$

Consider the all the tuples $(z_1, \ldots, z_n)$ with $z_1, \ldots z_k$ set to the distinct integers from 1 to k, and $z_{k+1} \ldots z_n$ each set to an arbitrary integer in the same range. Clearly there are $k!$ ways to select the first k elements and k^{n-k} ways to select the remaining elements. Each such tuple has exactly k distinct items, but it is not hard to see that if we attempt to continuously modify one of these tuples to equal another one, we must change the number of distinct elements, implying that each of these tuples is in a different connected component of Y.

To finish the lower bound proof, we argue that any instance of MSV can be reduced to the convex hull size verification problem (CHSV). Thus any lower bound for MSV problem applies to CHSV as well.

Theorem: The CHSV problem requires $\Omega(n \log h)$ time to solve.

Proof: Let $Z = (z_1, \ldots, z_n)$ and k be an instance of the MSV problem. We create a point set $\{p_1, \ldots, p_n\}$ in the plane where $p_i = (z_i, z_i^2)$, and set $h = k$. (Observe that the points lie on a parabola, so that all the points are on the convex hull.) Now, if the multiset Z has exactly k distinct elements, then there are exactly $h = k$ points in the point set (since the others are all duplicates of these) and so there are exactly h points on the hull. Conversely, if there are h points on the convex hull, then there were exactly $h = k$ distinct numbers in the multiset to begin with in Z.

Thus, we cannot solve CHSV any faster than $\Omega(n \log h)$ time, for otherwise we could solve MSV in the same time.

The proof is rather unsatisfying, because it relies on the fact that there are many duplicate points. You might wonder, does the lower bound still hold if there are no duplicates? Kirkpatric and Seidel actually prove a stronger (but harder) result that the $\Omega(n \log h)$ lower bound holds even you assume that the points are distinct.