A Survey of Convex Hull Algorithms

Adli Abu Shaban

April 30, 1997

1 Introduction

The convex hull of a set of points, Q, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. In other words, considering the points as nails sticking out of a board. The convex hull is the shape formed by a tight rubber band that surrounds all the nails. Figure 1 shows the convex hull of a set of points.

The convex hull of a set of points is simple yet very important data structure in computational geometry. It is one of the simplest shape approximation for a set of points. Many algorithms compute the convex hull as an initial stage in their execution, because convex polygons are often easier to deal with than points. For example to calculate the smallest geometric shape, like triangle or rectangle, it would be easier to calculate the convex hull first.

In this paper, I shall present few convex hull algorithms and compare them in terms of complexity, performance, and cost.

2 Naive algorithms

One might think to solve the convex hull problem by devising a simple algorithm. We can test if any given point lies within a triangle that is defined by

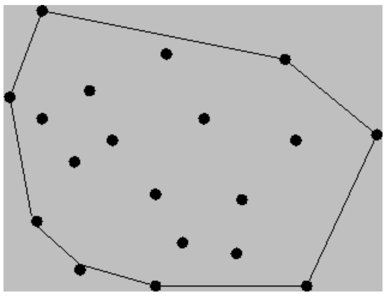

Figure 1: Convex Hull of a set of point in the plain

3 other points. We keep eliminating the internal points until we go through all possible triangles and eliminate all internal points. The remaining points would be the convex hull point. This algorithm takes $O(n^4)$ time, since there are $\binom{n}{3}$ triangle and each point has to be checked against each triangle.

Another algorithm would generate all possible convex hull edges and test whether all points lie to one side or another of this edge. If points lie on one side, then the edge is in the convex hull. Of course we have to take care of the case where points are collinear, that is three points exist in the same line. This algorithm runs in $O(n^3)$ time, since there are $\binom{n}{2}$ possible edges and it takes $O(n)$ time to check whether the points lie to one side or another.

3 Gift wrapping or Jarvis's march

Imagine we're trying to wrap a package in the plane, by wrapping some string or ribbon around it. Initially, we could tape one end of the string to some point which we know will be on the outside. (As in the algorithm for constructing a simple closed path, we could take the point with the lowest y-coordinate.) Then, we could pull the string taut (hold it off to the right), and wind it around until it hits another point in the point set. This point, too, will be on the convex hull. We can continue this way, winding the string around until it hits another point.

This procedure will eventually produce the convex hull of the point set. It remains merely to formalize this algorithm. I'll denote the points on the hull by h_1, h_2, $...h_M$, where M is the number of points which ultimately wind up on the hull.

Start off by setting h_1 to be the point with the lowest y- coordinate. This point is guaranteed to be on the convex hull; for if this point were not on the hull, since it does have the lowest y-coordinate, the segment between it and any other point would not be contained in the hull.

Again, if there are several points with the same lowest y-coordinate, we pick the one among them with the smallest x-coordinate. Subsequently, if we currently know the hull up to point h_i, we have to identify the next point as we continue winding the string around. This is equivalent to finding the point p_j with the minimum angle relative to h_i, provided that the angle between P_j and h_i is greater than the angle between h_i and h_{I-1} . This second condition is necessary, in order to not wind up going backwards in constructing the hull. Since we don't need the angle, merely the ordering relation it induces on the points.

Analysis Let M be the number of points which wind up on the convex hull, and N be the initial size of the point-set. For each of the M points, we have to find the next point on the hull. This involves finding the point with some minimum property, out of a set of size N. So the algorithm takes time $\Theta(MN)$ for each of the M points. If M is just a small constant, then this is good asymptotic behaviour. However, on average M grows with N. For example, if the points are randomly distributed inside a square, then $M = \Theta(\log N)$ on average, and the algorithm takes average $\Theta(N \log N)$ time. If the points are randomly distributed inside a circle, then $M = \Theta(N^{1/3})$ on average, and the algorithm takes average $\Theta(N^{4/3})$ time . Of course, in the worst case all of the points are on the hull. When this happens, M=N, and the algorithm takes time $\Theta(N^2)$.

4 Quick Hull

This algorithm is based on the quick sort algorithm. It runs in $O(n \log n)$ for best case and $O(n^2)$ for worst case. The worst case happens rarely, therefore this algorithm performs well on most cases. This algorithm makes use of the fact that most points lie inside the hull rather than on the hull and uses a method to eliminate the internal points.

Quick Hull begins by computing the points with the maximum and minimum, x- and y-coordinates. These points, of course, are hull points. Horizontal and vertical lines passing through these points are support lines for the hull, and therefore define a bounding rectangle, within which the convex is contained. All the points inside the quadrilateral defined by the 4 points could be eliminated since this quadrilateral is a partial hull. This process takes $O(n)$ time. The remaining points are classified into the 4 corner triangles that remain.

As the algorithm executes, we will have an inner convex polygon, and associated with each edge we have a set of points that lie outside that edge. The existence of these points means that this edge is not a convex hull edge. If there are no vertices lie outside the edge, then the edge is final convex hull edge. Next we find a point, c, among those outside-points, that maximizes the perpendicular distance from the edge, ab, that the points lie outside of.

We connect the two points of ab to c. Another technique for finding c might be the point that maximizes the angle ABC or cab. What we really need here is that point that tries to make an even split of the remaining points. Finally, we eliminate all the points that lie inside the triangle ABC. We get two more set of points; those that lie outside ac, and those that lie outside bc. We keep recurring until we find the hull points.

Analysis The running time of QuickHull, as with quick sort depends on how evenly the points are split at each stage. If we let $T(n)$ denote the running time on the algorithm, where the n is the number of points that remain in the current area, then the time is give by the recurrence relation:

$$T(0) = 1$$

$$T(n) = cn + T(n_1) + T(n_2),$$

where n_1 and n_2 are the number of points remaining in the two areas as a result of the triangle split. As the QuickSort, the running time of this algorithm is good as long as $\max(n_1, n_2)$ is not too close to n.

5 Graham's Scan

The idea is a common one in computational geometry, known as an incremental algorithm: add items one at a time to some structure, maintaining as you do a partial solution for the items added so far. The order in which items are added is often important. The Graham scan adds points in sorted order from left to right, according to their x-coordinates. So $O(n \log n)$ time is needed for an initial sorting stage, or less time if you want to assume e.g. that the points' coordinates are small integers.

The algorithm is easier to describe in terms of the upper and lower chains. I will describe how to compute the upper chain; the lower chain is completely symmetric. Because of the order in which we add points, the first point is always the left end of the upper chain, and the last point is always the right end. We know it should be at the right end, so let's try adding it after the previous rightmost point. However, this might make an indentation at that previous point; if so we remove the indentation from the chain, making it have one less edge. We keep removing indentations one by one until there are none left, at which point we have the new chain.

This is easiest to implement if we maintain the chain using a stack data structure. Each new point in the chain corresponds to a push in the stack, and removing an indentation corresponds to a pop. Here is some pseudocode:

stack S = 0. for each point (x,y) in sorted order by x if (x,y) and top two points on S are indented pop S push (x,y) onto S

How do we test whether three points (x0,y0), (x1,y1), and (x2,y2) form an indentation. Since we know in this case the ordering of the x-coordinates of the three points, all we need to do is test whether the middle point is above or below the line segment formed by the other two. The remaining figures on this report show the Graham Scan algorithm.

Analysis The algorithm goes through $O(n)$ iterations of the outer loop. There is also an inner nested loop, but each iteration of the inner loop removes a point from the stack, and each point can only be removed once, so this happens $O(n)$ times total. Therefore the total time for the convex hull algorithm, after the initial sorting stage, is $O(n)$. The overall algorithm takes time $O(n \log n)$ because of the sorting step.

A similar process of shortcutting indentations also works for finding convex hulls of simple polygons. Of course you could just sort the vertices of the polygon and apply Graham scan, but you can instead do the shortcutting process on the polygon itself. It is possible for a shortcut to add a crossing between two edges, but these will always become untangled by

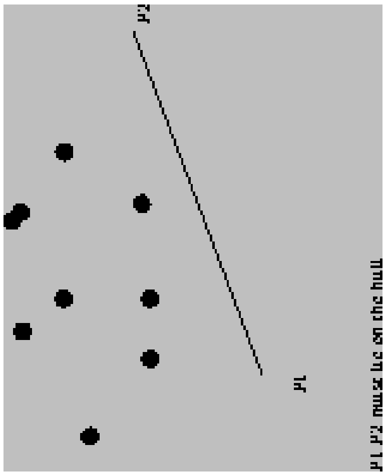

Figure 2: Sort the points about the lowest point

later stages of the angles. In this way, one can find convex hulls of simple polygons in linear time, without taking the time for a sorting stage. The Graham scan algorithm has been implemented many times in different ways and languages.

6 Divide and conquer

This algorithm is based on the mergsort algorithm and it takes $O(n \log n)$ time to calculate the convex hull. This algorithm works by sorting all the points by the x-coordinates. Then it divides the points into two sets A and B, where each of A and B contains half the point set. The algorithm works by recursively computing $H_A = conv(A)$ and $H_b= conv(B)$. The hulls, are merged finally into a common convex hull which is the final hull.

The recursion stops when either A or B contains three or fewer points in which case it is easy to compute the hull in $O(1)$ time.

Analysis The sort takes $O(n \log n)$ time. The running time of the actual algorithm, excluding the sort, is give by the following recurrence relation.

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

The recurrence solves to $T(n)$ if $f(n) \in O(n)$. We need to get the two convex hulls to merge in time $O(n)$. One thing that could help simplify the algorithm is the fact that the two hulls are always separated by a vertical line. Therefor, merging the two hulls is done by computing the two tangent lines. An upper tangent and a lower tangent. Finally we eliminate the points that lie between these tangents.

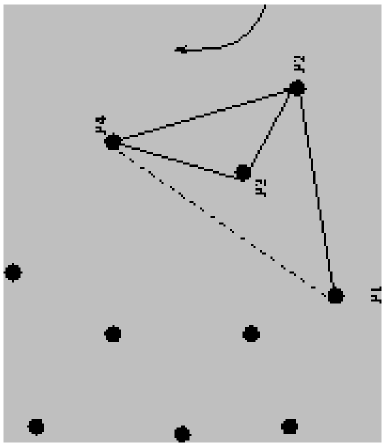

Figure 3: Two points p1 and p2 are on the hull

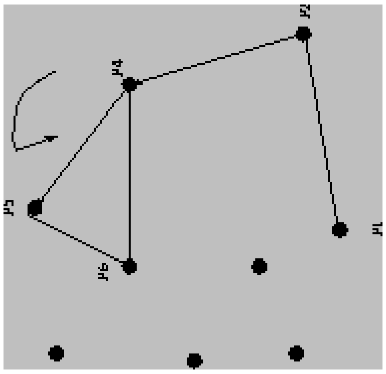

Figure 4: A point, p3, lies inside the hull and is eliminated

I will present how the lower tangent is computed, the upper tangent is similar. The algorithm works by going through all the potential points. First, it initializes a to be the right most point of H_A and b to be the leftmost point of H_B. Finding these points takes a linear time. Lower tangency is a condition that can be tested by an orientation test of the two vertices and the neighbouring vertices on hull. Second, it iterates the following two loops, which walks from a to b until reaching the lower tengency point.

Lower Tangent

1. Let a be the right most point of H_A.
2. Let b be the left most point of H_B.
3. While ab is not a lower tangent for all H_A H_B do

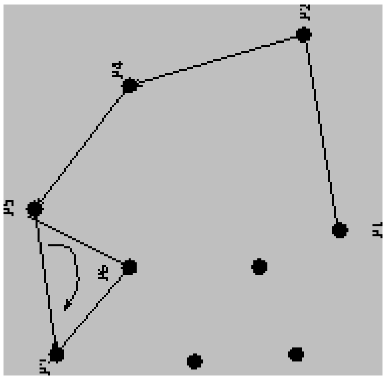

Figure 5: Building the convex hull, no elimination

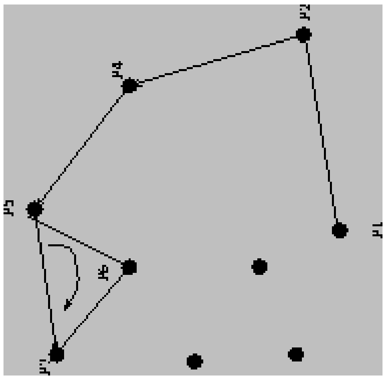

Figure 6: Elimnate a point

   (a) while ab is not a lower tangent to H_A do $a = a - 1$.

   (b) while ab is not lower tangent to H_B do $b = b + 1$;

4. return ab.

7 Ultimate Kirkpatrick and Seidal's algorithm

This algorithm idea is to develop a smart divide and conquer algorithm that runs in time $O(n \log h)$ where h is the size of the hull. Therefor, this algorithm is an improved divide and conquer algorithm.

We, first, divide the points in S into two sets S_L and S_R of approximate equal size by using a vertical separating line. We can compute the upper supporting line before we compute the convex hulls of S_L and S_R, since these are functions of the point set and not of the convex hulls of S_L and S_R. We can delete all points that are immediately below the bridge. If two

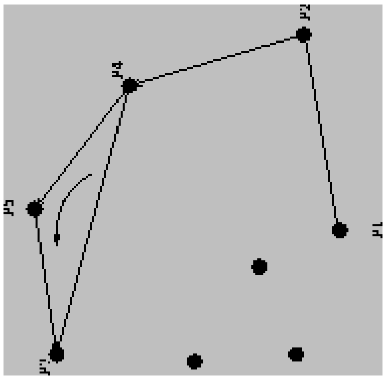

Figure 7: Go back and eliminate more than one point

points, say p and q, are the bridge. We can delete all points with x coordinates between p and q. We keep doing the same thing recursively on the remaining points to find the upper chain of S'_L and S'_R. This will reduce the number of points we recur on.

This algorithm does not presort the points before starting the algorithm. At each divide step we have to invoke a linear time median finding algorithm to split S into equal pieces. Here, we could use a randomized algorithm to find an approximate median quickly.

We use a method to find the bridge in $O(n)$ time using a method called prune and search. This method spends cn time looking for points to delete and recurring. The running time for this is

$$cn + c\ n/2 + c\ n/4...c = O(n).$$

Analysis Starting with h =1, the case when the first permanent hull point is found, we compute a bridge in any recursive call and we find that the bridge connects leftmost and rightmost points. The remaining points can be eliminated immediately. This will take $O(n)$ time. If The upper chain of S'_R has h_R and the upper chain of S'_L has size h_L, then we have that $h = 1 + h_R + h_L$. The running time of the algorithm can be expressed as $T(n, h)$.

$$T(n, h) = cn\ if\ h = 1.$$

$$T(n, h) = cn + T(\frac{n}{2}, h_L) + T(\frac{n}{2}, h_R).$$

We need to argue that regardless of the split of h between h_L and h_R, the algorithm runs in $O(n \log h)$ time. This can be proved by induction. Assume $T(m, h') \leq cm\ \log\ h'$ for $m < n$ and $1 < h' < h$.

$$T(n, h) =\leq cn + c\frac{n}{2} \log h_L + c\frac{n}{2} \log h_R.$$

Since $\frac{1}{2}(\log h_L + \log h_R) \leq \log \frac{h_L+h_R}{2}$, then

$$T(n,h) \leq cn + c\frac{n}{2}(2\log(\frac{h_L+h_R}{2})) \leq cn + cn(\log h - 1) \leq cn\log(h).$$

Lets know try to compute the bridge. We start by assuming that the bridge has slope $K*$. We will come up with a value for the slope K and try to figure out how $K*$ compares to K. Through each point p_i in S. Let the line slope be l_i. The line l_i which has all the points of S below it is the line we are looking for. This line has the highest intercept with the dividing line. In $O(n)$ time we can compute which line has this property. If the line is defined by a point from S_L and a point from S_R this is the bridge we are looking for. If the line is defined only by points of S_L, then $K* < K$. If the line is defined only by points S_R then $K* > K$.

To apply the search, we need to discard a constant fraction of the points to continue looking for the bridge. We might come up with a correct slope. At that time the algorithm stops. If the guess is not right, we recurs on a constant fraction of the points. The running time is a decreasing geometric series and converges to $O(n)$. We now outline the search method. Consider a pair of points $a, b \in S$. Assume that a is to the left of b. Let the slope defined by this pair of points be K'. Suppose $K* > K$ and $K \leq K'$, then $K* > K'$. In this case we claim that b cannot be a bridge vertex. In either case, we are able to eliminate points as being candidate vertices.

How do we guarantee that we can find sufficiently many lines that have slope less than K and many lines with slop more than K. We arbitrarily pair up the points in set S, and compute their slope. We, then, select the median slope of this set of $n/2$ lines, and use that as our guess for the slope of the bridge. We either guess the slope correctly and stop, or in either case eliminate one point from $n/4$ pairs. We are left with at most $3n/4$ points and recurs on this set of points. This completes the search algorithm.

8 Randomized Incremental algorithm

Incremental convex hull algorithms is relatively simple to implement, and generalizes well to higher dimensions. Gift wrapping algorithm, presented earlier in this report, is often used in higher dimensions, but it's worst case performance is not good even in the plane. This algorithm works by inserting points on at a time and incrementally updating the hull. As with every randomized algorithm, the expected running time depends on the random order of insertion, but it is independent of the point set. There was an attempt by O'Rouke to do a deterministic a version of this algorithm by inserting the points according to their x-coordinates. The worst case of the deterministic algorithm is $O(n^2)$ in three spaces, where the randomized version's achieves $O(n \log n)$.

The algorithm works as follows:

1. Permute the points randomly
2. Let $P = (p0, p1, ..., p_{n-1})$ be a sequence.
3. Let H_2=conv(p0,p1,p2) be a triangle.
4. For I running from 3 to n-1, insert p_i into the existing hull.

Here, we assume that each point has yet to be inserted has a pointer to one edge of the current convex hull such that this point lies outside of this edge. If the point lies inside the hull, then it is marked as inside and no further processing is needed. One way to consider this association is to consider some point K that lies in the interior of the convex hull. Imagine a line segment drawn from each point p_i to K. This segment must hit some edge of the hull. We associate p_i with this edge. In addition, each hull knows which point associated with it. For each edge e, let $L(e)$ denote the set of points associated with e in this way, and let $l(e)$ denote the size of this list.

When a new point is added to the convex hull, H_{I-1}, to form the new hull H_i, we follow p_i's pointer to the edge that p_i lies outside from. We walk clockwise and counterwise around H_{I-1}. We connect p_i to these points, and discard the edges that were traversed from the hull. The resulting polygone is H_i. For each each hull edge, e, that was discarded, we access each of the points in $L(e)$ and determine whether if it now lies outside of (p_a, P_i) or if it lies outside of (p_i, p_b) or if it does not lie outside any of them. In the first case, the point is transferred to $L(p_a, p_i)$, in the second to $L(p_i, p_b)$ and in the third it is marked as inside because it lies within the current hull.

Analysis In the analysis of this algorithm, we have to take care of three things; destruction of points, creation of new edges, reclassification of points according to the edge that they now lie outside of. Since each edge that is destroyed should have been created before, thus the time spent destroying edges can't be much different from the total time spent creating edges. Since the algorithm creates 2 edges for each of $n - 3$ stages, the total time spent creating and destroying is edges is $O(n)$.

The only thing left to analyse is the expected time of reclassification. We claim that the total reclassification time will be $O(n \log n)$ in the expected case. Reclassification can result in a point being labelled as inside. However, this happens only once for each point in the interior of the final hull. Therefore, the total cost for these reclassification cannot exceed n.

Let us concentrate on counting the expected number of reclassification of points that remain outside the current hull. Consider the last point to be added into H_i With equal probability, $1/I$, each point in $p_0, p_1, ..., p_i$ was the last to be added. Let p be this point. If p is interior to the current hull, then no work was needed in reclassification. Otherwise, p will be on the current hull, H_i. Let e_1 and e_2 be the two edges incident to p on H_i. The uninserted points that would have to be reclassified, are just the points that are currently in $L(e_1)$ and $L(e_2)$. Therefore, the cost of reclassification would be $l(e_1) + l(e_2)$ if p were the last to be inserted. To determine the expected cost, we sum over these values over all I points, and multiply by the probability $1/I$ that this was the last point added. SO for each edge e, $l(e)$ is added twice, once for the left point and once for the right point. The expected cost of reclassification in the construction of H_i

$$\frac{1}{I} \sum_{e \in H_i} 2l(e) = \frac{2}{I} \sum_{e \in H_i} l(e).$$

Since the sum of $l(e)$ counts each of the remaining $n-I$ points at most twice. This sum evaluates to $2(n-1)$ and thus the expected cost of reclassification is.

$$\frac{2(n-I}{I} \leq \frac{2n}{I}.$$

Summing up all stages to get the total time of reclassification

$$T(n) = \sum n-1^{I=3} \frac{2n}{I} = 2n \sum n-1^{I=3} \frac{1}{I} \approx 2n \ln n \in O(n \log n).$$

9 Remarks and conclusion

This report outlined the 2-dimension convex hull algorithms that were mentioned in the literature. These algorithms range from easy to implement and understand, takes long time to execute, hard to implement. Therefore, we have to trade off these properties as we are selecting an algorithm. I will try to present these trade offs later but first I would like to discuss a lower bound limit that no convex hull algorithm could perform better than.

In David Mount's notes, a claim was made that a given convex hull algorithm can not perform better than $O(n \log n)$. He starts by defining two versions of the convex hull problem. The first one is to compute the sequence points lying on the convex hull in cyclic order about the hull. Second one is compute the set of points on the convex hull, without regard to order.

The first problem seems to be the more natural statement of the convex hull problem. And it is easier to prove that this problem requires $O(n \log n)$ time. However because its output appears in sorted order, it is natural to ask whether the $O(n \log n)$ is needed to determine the ordering and if we can compute the points of the hull without ordering them. It turns out that we can not compute the hull without ordering the points. Since we know that no sorting algorithm can perform better than $O(n \log n)$. Therefore, the convex hull takes at least $O(n \log n)$. The best convex hull algorithm is the one that minimizes the time of the algorithm to the $O(n \log n)$ limit.

The idea of the proof is to reduce the sorting problem to the problem of computing a convex hull.

Suppose we want to sort a set of numbers $x_1, x_2, ...x_n$. First, we compute the (x_i, x_I^2) for $1 \leq I \leq n$. We notice that every point of the first step is on the convex hull. Second, we compute the hull by any desired method. Third, we determine the point with the lowest x-coordinate in linear time. We walk counterwise around the hull and we notice that the x-coordinates of the points give the sorted order of the original points. We can notice that the time for transformation and sorted order extraction are $O(n)$. We know that the current procedure can not run faster than $\Omega(n \log n)$.

In selecting a convex hull algorithm, one wants an efficient, easy to implement and understand, and quick algorithm. If we are going for easy algorithms, pick the naive algorithms. Naive algorithms perform extremely bad when the number of points gets higher. Since most of the real life problems, that needs to calculate the convex hull, have large number of points, naive algorithm should be avoided for most problems.

For some of the presented algorithms, the performance gets bad when the number of points on the hull increases. Although it is not normally the case, but in the worst case all points could be on the hull. Since this not normally the case, some application would have so little number of points on the hull. In that case, some algorithms, like gift-wrapping achieves $O(nh)$, where h is the number of hull points. This is good and beats the lower bound time discussed earlier. However, the performance of the gift-wrapping can get as bad as $O(n^2)$. Our aim is to achieve the $O(n \log n)$ for the worst case, from some algorithm.

Gift wrapping algorithm is easy to implement. However it can only be quick if the points of the hull are in a certain shape. The worst case performance of the gift wrapping is $O(N^2)$. I would say that the average shape of a convex hull is a set of points that are randomly distributed inside a circle in which case the gift wrapping algorithm takes $O(N^{4/3})$. Which a little bit slow.

Since QuickHull is based on the QuickSort algorithm, it runs in $O(n \log n)$ in the best case and $O(n^2)$ in the worst case.

Divide and conquer algorithm is based on the merge sort algorithm and takes $O(n \log n)$ time to calculate the convex hull. It works by dividing the problem into to two parts and solving the hull. Finally, it merges the hulls. The idea is simple and it takes good time to compute the hull.

The Ultimate algorithm takes the least time to compute the hull, $O(n \log h)$, where h is the size of the hull. However, it is not easy to implement. In some case there no notable time between $\log h$ and $\log n$, which is not worth the extra efforts

Randomized Incremental algorithm is relatively simple to implement. It works by randomly inserting points into the partial hull and updating the hull. This algorithm takes $O(n \log n)$ time to compute the hull. I recommend this algorithm for the convex hull problem.

References

[1] David Mount. Analysis of Algorithms course notes, spring 1994.

[2] Samir Khuller. Analysis of algorithms course notes, spring 1996.

[3] Anil Maheshwari. Computational Geometry course notes at Carleton University. Winter 1997.

[4] D. G. Kirckpatrick and R. Seidel, SIAM Journal of Computing, Vol. 15, 1986, 287-299.

[5] Roberto Tamassia. Convex Hull. http://wilma.cs.brown.edu/courses/cs016/packet/

[6] C. B. Barber, D. P. Dobkin. The Quickhull Algorithm for Convex Hull. http://www.geom.umn.edu/docs/preprints/online/GCG53.htm.

[7] Jeff So. Convex Hull. http://www.cs.umd.edu/ dclipper/hull/ch.html.

[8] J. Richard Bradley and Steven S. Skiena. 1.6.2 Convex Hull. http://www.cs.sunysb.edu/ algorith/files/convex-hull.shtml