Empirical Analysis of Convex Hull Algorithms
CS 350
Fall 2016
Neil Schlunegger
11/28/2016

1. Introduction

The purpose of this paper is to analyze three separate algorithms for solving the convex hull problem. In order to understand the problem at hand, a few definitions have to be established. A convex set is a region such that, for every pair of points within the region, every point on the straight line segment that joins the pair of points is also within the region[1]. The convex hull of a set X of points in the Euclidean plane is the smallest convex set that contains X[2]. A more informal understanding of a convex hull can be attained by taking a rubber band and stretching so that it encloses all of the points of X. The outlines of the rubber band are the boundary of the convex hull. Real-world applications of convex hull include computer visualizations, path finding for mars rovers, geographical information systems, among many others[3]. Although certain convex hull algorithms may be used for arbitrary dimensions, for the purposes of this paper only 2 dimensional space will be used

2. Algorithm Descriptions

Quickhull[4 5]

The first algorithm used in this paper will be the quickhull algorithm, or simply quickhull for short. Quickhull begins by taking the leftmost (A) and rightmost (B) points of a passed set of points, and adds those points to a temporary set of points which will later be returned as the entire convex hull. The added points are consequently removed from the original set of points. The reasoning for this operation is that the leftmost and rightmost points are guaranteed to be part of the convex hull.

```
for (int i = 0; i < originalPoints.size(); i++){
    Point p = originalPoints.get(i);
    if (pointLocation(A, B, p) == -1)
        leftSet.add(p);
    else if (pointLocation(A, B, p) == 1)
        rightSet.add(p);
```

Figure QH1(left) – determining if point p is on the right or left side

```
public int pointLocation(Point A, Point B, Point P)
    int val = (B.x - A.x) * (P.y - A.y) -
              (B.y - A.y) * (P.x - A.x);
    if (val > 0) {return 1;}
    else if (val == 0){return 0;}
    else {return -1;}
```

Figure QH2(right) – positive values are on the right and negative on the left

If one were to draw a line between the leftmost and rightmost points (AB) then all point candidates for the convex hull would fall to the right or to the left of said line. This algorithm makes use of this fact, so points are divided up according to which side of the line they fall on. If the point falls on the line segment between the leftmost and rightmost point, then the point is ignored because it is already enclosed by A and B and cannot be part of the convex hull. Once the points are separated, a recursive auxiliary function is called to determine which points are part of the convex hull – one root function with the set of points to the left of the line, and one for the right of the line. Both get passed points A and B, the left set(or right set), and set of points that contain the "hull so far", which upon the first call is simply the leftmost and rightmost points.

```
hullSet(A, B, rightSet, convexHull);
hullSet(B, A, leftSet, convexHull);
```

Figure QH3(left) – Recursive calls on the line segment AB, the right + left halves of AB, and the hull so far.

```
public void hullSet(Point A, Point B, ArrayList<Point> originalSet,
                    ArrayList<Point> hull)
```

Figure QH4(right) – Header of the auxiliary function

The purpose of this auxiliary function is to suss out the remaining points of the hull. It does so by first determining which point is the furthest away from A and B, called point P. Point P is part of the convex hull because there is no point that will enclose it in the hull, so it is added to A and B as part of

the “hull so far” and removed from the original list of points. Now, because we have 3 points of a triangle that enclose some other points, we can safely remove all points that lie within this triangle, because by definition they cannot be part of the hull. We do this implicitly through a few steps. Step one is to create a list of points that lie to the left of segment AP. Step two is to create a list of points that lie to the left of PB. LeftSetAP & LeftSetPB represent these sets.

```
hullSet(A, P, leftSetAP, hull);
hullSet(P, B, leftSetPB, hull);
```

Figure QH5(left) – recursively calling the auxiliary function hullSet

```
int insertPosition = hull.indexOf(B);
if (originalSet.size() == 0)
    return;
if (originalSet.size() == 1){
    Point p = originalSet.get(0);
    originalSet.remove(p);
    hull.add(insertPosition, p);
```

Figure QH6(right) – base case of hullSet

The third step deals with the other side of the triangle and is dealt with by calling this auxiliary function again within the quickhull algorithm as shown in figure QH3. All points that are not within these new lists can simply be ignored because they are enclosed and thus not part of the hull. Once these lists are populated, this auxiliary function is recursively called and keeps creating triangles to eliminate and locate points by following the same process as above, until the base case in QH6.

Space Complexity of Quickhull

Quickhull takes in a list of points of size n. Another list is created of size n which contains the convex hull of the input points. There are other variables and temporary lists created within the algorithm; however, they may only encompass anywhere from 0 up to n – 2 points depending on the point distribution on the sides of A and B. Therefore the list of the points of the hull is the largest consumer of space, which means this algorithm takes O(n) space.

Time Complexity of Quickhull

When there are no more points, there are no more operations to perform, so T(0) = 0. Each recursive call is done after taking out one point from the original set of points, so T(n) = T(n-1). But the entire set of points must be scanned n times in the worst case in order to find the extreme point that is used for triangulation, so in total the recurrence relation is:
T(n) = T(n-1) + (n) //(1)
T(0) = 0
Now, T(n-1) = T(n-2) + (n-1) //(2)
And, T(n-2) = T(n-3) + (n-2) //(3)
Also, T(n-3) = T(n-4) + (n-3) //(4)
Substitute (2) into (1) to get T(n) = T(n-2) + (n-1) + (n)
Substitute (3) into the above line to get T(n) = T(n-3) + (n-2) + (n-1) + (n)
Substitute (4) into the above line to get T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + (n)
Put into a general formula, this is: T(n-i) + (n-(i+1)) + … + n
Substituting i = n, this becomes T(0) + 1 + 2 + … + n
But T(0) = 0, so this is the classic summation 1 + 2 + … + n

$$\sum_{i=0}^{n-1} i = \frac{1}{2}(n-1)\,n$$

$\in O(n^2)$

However, if we are assuming random distribution of points, the worst case is unlikely to happen, so it will behoove us to discuss the average case. By splitting into left and right halves, we are

using a divide and conquer technique.

Dividing all of the points will take n amount of time. Assuming random distribution, we can suppose that an equal amount of points will fall on each half of the original AB line segment. And we are calling the auxiliary function twice recursively, for 2 * T(n/2) time complexity. When we are determining which points to use within the auxiliary function, the rest of the n points are examined. The recurrence relation can be stated as: T(n) = 2T(n/2) + n (with T(0) = 0 as before). Using the master theorem, a = 2, b = 2, d = 1.
$a = b^d$, so in the supposed average case, this algorithm $\in \theta(n \log n)$

Graham's Scan[6 7 8]

The second algorithm used in this paper is Graham's Scan. This algorithm keeps track of the points of the convex hull through a stack. To start out, this algorithm sorts the set of points by their polar angles in counterclockwise (ccw) order, in reference to a horizontal line drawn through the leftmost point with the smallest y-coordinate, called k0. k0 is pushed onto a stack, then the smallest polar angle in reference to k0, called k1 pushed onto the stack as well. At this point begins a for-loop starting at the index of the 3rd point in the sorted list, called k2.

```
for (int i = k2; i < N; i++){
    Point top = hull.pop();
    while (Point.ccw(hull.peek(), top, pts.get(i)) <= 0)
        top = hull.pop();
    hull.push(top);
    hull.push(pts.get(i));
```

Figure GS1 – looping through the remaining points. Point.ccw uses the same formula as pointLocation of figure QH2

At the beginning of the for-loop, the top of the stack is popped off and stored in a variable called top. Then, a nested while-loop is called that compares the point at the index of the for-loop with the variable top, and the element in the stack that comes after top, to check which direction the next point is relative to the current top of the stack. This while loop will continually update the variable top as the next element in the stack until the next point to be examined makes a right turn from the variable top. This whole process is done because if the current index is part of the hull, it will make a right turn from the current top, so the current top will be enclosed in the hull by the current index, thus the current top will not be part of the final stack of points. After the while loop is exited, the variable top is pushed onto the stack, followed by the index in the for-loop. Then, the loop begins again at the next index. This process is repeated until the entire set of points is traversed in ccw order.

Space Complexity of Graham's Scan

There are a few small variables that are calculated, but the dominating piece of space in this algorithm is the stack that is used. This stack may include all of the points of the original list of points in the worst case, so this algorithm uses O(n) space.

Time Complexity of Graham's Scan

First, the list of points is sorted. Then it is sorted again by polar coordinates. According to official documentation, Collections.sort takes O(nlogn) time.[9] There is some additional setting of indices that happen before the for-loop, but these operations should not usually take more than a very small constant amount of time if there are not a lot of duplicate points. The for-loop will go from the 3rd index all the way until the last index of the set of points. Its basic operation is the direction check which happens as a precursor to entering the while-loop. This direction check is done via simple vector arithmetic, and so it is dominated in time by the outer for-loop and does not add to the greater complexity. The for loop has complexity O(n), and thus is dominated by the initial sorting of O(nlogn).

Therefore this algorithm ∈ O(n log n).

Monotone chain[10] [11]

The monotone chain is the third and final algorithm we will analyze. It is called such because this algorithm computes the upper and lower hulls of a monotone chain of points. In math terms, monotonic means a function is entirely increasing or decreasing.[12] The upper hull is constructed using an increasing monotonic set of points with respect to the x-coordinate, and the lower hull with a decreasing set of points. This algorithm is similar to Graham's scan in that it presorts the points, and that it compares those sorted points to determine which is part of the hull.

```
for (int i = 0; i < n; ++i) {
    while (k >= 2 &&
           cross(hull.get(k-2), hull.get(k-1), originalSet.get(i)) <= 0)
        k--;
    hull.set(k++, originalSet.get(i));
```

Figure MC1 – building the lower hull. Cross uses the same formula as pointLocation shown in QH2

As stated above, the monotone chain starts out by presorting the points. Then begins the building of the lower hull, which runs through all of the points in a for-loop. There is a nested while-loop which will set the index (called k) of the point which is to be added to the lower hull, and will run so long as the index doesn't make a ccw turn and there are still at least 2 points in the lower hull. But for the first couple of points, the while loop is skipped in order to start building the chain. After the lower hull gets index k added, k is increased and the next for-loop iteration begins. This k index is also used in the while loop to determine which way the next index of the main set of points turns. The upper hull is created similarly, but with the index i starting at the other end of the set of points counting downwards, and a second index t, which starts sequentially one index after where k left off, and is used to join the lower and upper hulls together. Afterward, the points that were duplicated, k and k-1, are removed and a list is returned which contains the hull.

Space Complexity of Monotone Chain

As in the Graham's scan algorithm, there are a few variables that are kept track of, but they are dominated by the list that contains the hull. This list may possibly contain all of the original set of n points. In the Java implementation, there was trouble creating a sublist without creating a new empty list first, then copying the sublist of the original points into that new list, so there is another n space being taken to go through the points creating the temporary list. But overall, n + n = 2n is still simply O(n).

Time Complexity of Monotone Chain

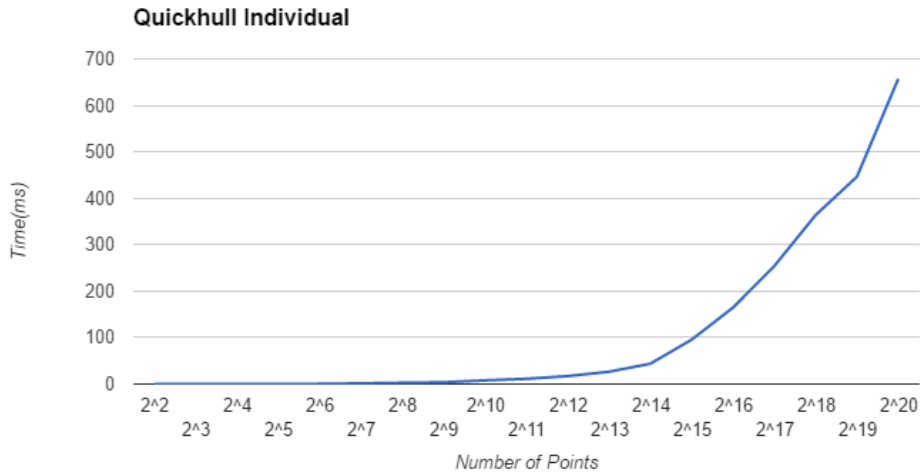
Collections.sort is used to sort in this algorithm, which as we know has O(nlogn) time complexity[9]. Building the lower and upper hulls both iterate through the entire set of points, for n + n = 2n time. Calling the .sublist function will also take n time to go through all of the points. Overall, nlogn + 3n is simply ∈ O(n log n).

3. Experimental Procedure

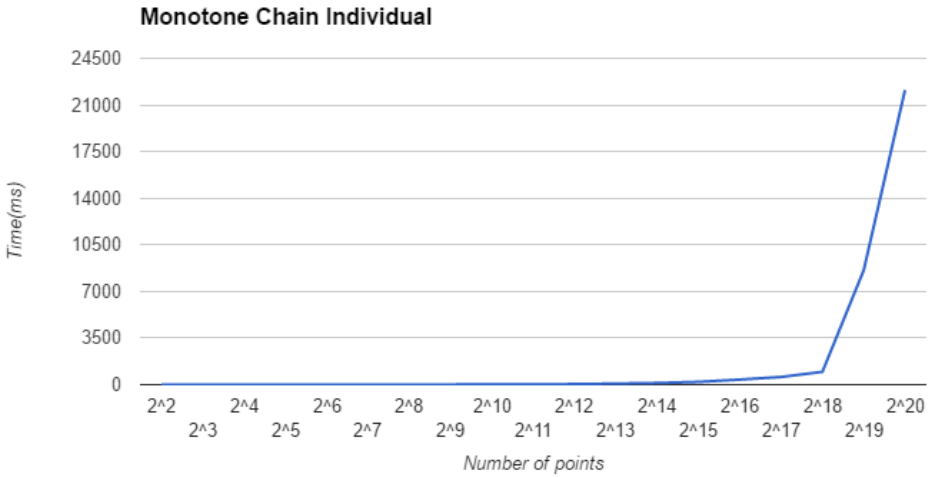
The three algorithms were implemented and tested on an Lenovo Thinkpad E-540 with Java 8 using Eclipse Mars version 4.5.1. A 64-bit Windows 10 system that has an Intel i5-5200U CPU @ 2.20 GHz and 7.74 GB of free RAM. The only program that was run at the same time as testing was Google Chrome version 54.0.2840.99 with a Google spreadsheet open.

Creation of points was done outside of the calling of the algorithms. Random points were used, ranging in value from -20,000 to 20,000 (inclusive) for both x and y-coordinates. Memory for variables was declared outside of the scope of algorithm testing so that there was no interference with the timing. Algorithms were tested individually with their own set of random points 10 times, and averaged out. Algorithms were also tested together and used the same (but a different set from above) set of random points 10 times, and averaged out. Testing was carried out on a number of points ranging from 2^2 up to and including 2^{20} to visualize how each algorithm handed a growing number of points.

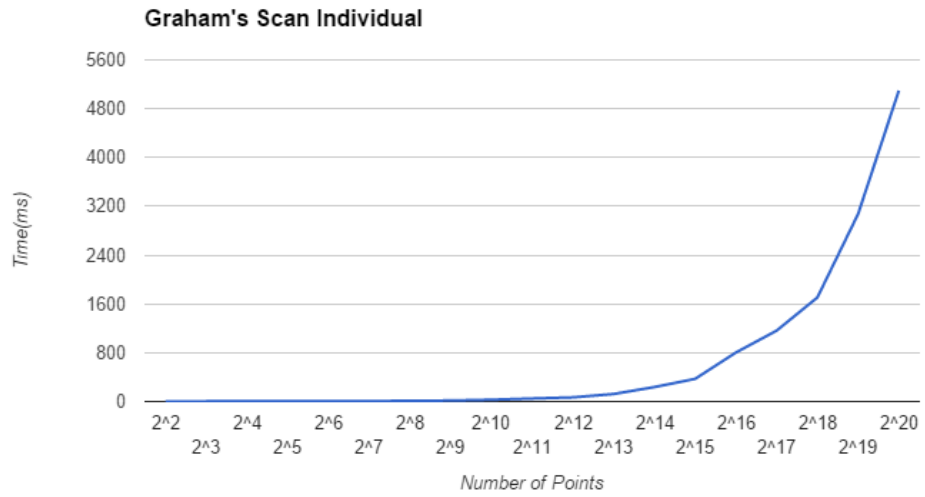
4. **Experimental Results**



Graph 1 – Individual Quickhull

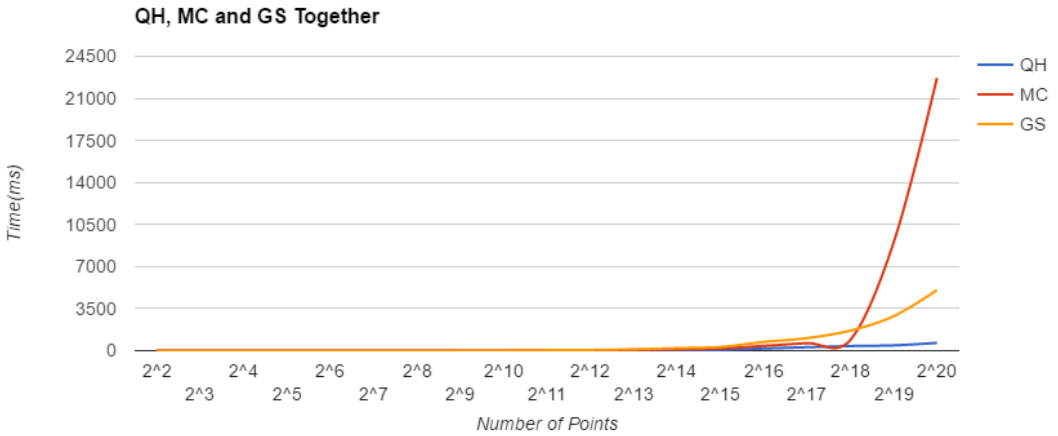


Graph 2 – Individual Monotone Chain



Graph 3 – Individual Graham's Scan

As evidenced by the shape of their functions, each algorithm obviously takes more time based on how many points it has to deal with. These graphs were not scaled the same on the y-axis above in order to better illustrate the shape of their curves. For smaller inputs, there is not much difference in the time taken to compute, but as the input keeps growing the incline gets much sharper for the monotone chain. This is not something to be expected given their respective complexity classes; however, big-O notation does not take into account the full breadth of the space and time an algorithm uses because it discards dominated constants and does not take into account how an actual processor may handle operations. These costs show up in the above graphs.



Graph 4 – Quickhull, Monotone Chain, and Graham's Scan on the same points

The three algorithms were also tested on a randomized set of points that was the same for each algorithm in order to visualize their differences better. The rates at which the monotone chain and quickhull algorithms grow in real terms can be better observed.

5. Conclusions

In order of difficulty to understand and implement, I found quickhull to be the toughest, followed by monotone chain, with Graham's scan being the easiest. Quickhull was the hardest because

of the helper function and recursion it used. Monotone chain fell in the middle, but it was very similar to Graham's scan. When I was going through the algorithms by hand and tracing out how they formed their respective hulls, monotone chain made the most sense to me intuitively because of its simple looping through all of the points

It was interesting to play around with these algorithms with smaller point sets and print out the hulls and the small amount of time it took to calculate them. One interesting thing I noticed that I didn't analyze, was that as I made the point sets more dispersed (e.g. range from -100,000 to 100,000) the quickhull could take the same amount of time, or even be slower than the other two algorithms. I chose to use an arbitrary range of -20,000 to 20,000 for simplicity's sake, but the end results would have been different had I opted for a wider range.

One thing that I would like to do in the future is look at using the applications of these hulls. The most interesting use I found when researching this project was that the Mars rover uses convex hulls to explore the surface of Mars. Also, using 2-dimensional space was a fine exercise, but for robotic purposes it would be interesting to look at algorithms for 3-dimensional spaces and how robots use them.

References

1. https://en.wikipedia.org/wiki/Convex_set
2. https://en.wikipedia.org/wiki/Convex_hull
3. http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm
4. http://www.ahristov.com/tutorial/geometry-games/convex-hull.html
5. https://en.wikipedia.org/wiki/Quickhull
6. http://www.sanfoundry.com/java-program-implement-graham-scan-algorithm-find-convex-hull
7. http://algs4.cs.princeton.edu/99hull/GrahamScan.java.html
8. https://www.youtube.com/watch?v=QYrpHE8iDGg
9. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html
10. http://wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain
11. http://geomalgorithms.com/a10-_hull-1.html
12. https://en.wikipedia.org/wiki/Monotonic_function