Portland State University

CS350

Algorithms And Complexity

ConvexHull Analysis: BruteForce vs. QuickHull

Author:
Wes Risenmay
Josh Willhite

Instructor:
Dr. Andrew Black

March 12, 2014

Contents

1 Objective

The purpose of this paper is to compare convex hull algorithms specifically, brute force vs. QuickHull. Our comparison will consist of an analytic time efficiency analysis of the two algorithms followed by actual timing of the algorithms for the same input data. We'll conclude with an explaination of how our predicitions compared to the data.

2 ConvexHull Overview

Given a set of points in the XY plane, imagine stretching a rubber band around the set. All of the points that the rubber band touchs are on the convex hull *[Levitin pg.110]*.

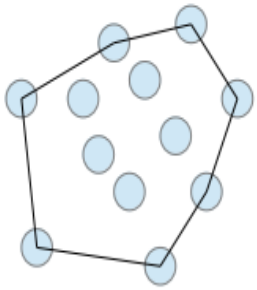

Figure 1: Convex hull Example

Finding the convex hull of a given set ends up being critical in a bunch of different areas, from computer vision to Geographic Information Systems to CAD applications *[Levitin pg.110]*

3 Algorithm Explaination

3.1 Brute Force

The brute force approach to finding the convex hull of a set of points is to use 3 nested loops. The two outer loops get a point(a) and point(b) which form a line. The inner loop than calculates the determinant of of the two points which make up the line with a third point(c). If two of the "c" points are ever on opposite sides of the line (determinant has different sign) than we know line(a-b) can't be on the hull and we move on.

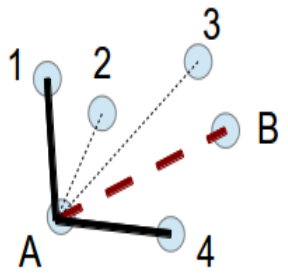

Figure 2: Brute Force Example

In the above diagram our the line we're testing would be from A-B. First AxBx1 would be calculated, then AxBx2 would be compared to the first determinant they're the same so the algorithm would continue until AxBx4 is reached. Since the sign of this last determinant does not match the other the line A-B is not on the hull and we can move on to the text the next line.

3.2 QuickHull

The QuickHull approach is both more elegant and more performant. (1) Find furthest left and right points A,B in the figure below. These two points are guaranteed to be on the hull. (2) Split the remaining points into two subsets above/below the base edge. (3a) The recursive step starts by finding the furthest point from the base edge, in the below figure it's labeled "Pivot" for the top subset. (3b) Now divide the current set of points into two subsets consisting of points to the right of the line from Pivot-¿B and points to the left of the line from Pivot-¿A. (3c) Now call the recursive function twice again, once for the base edge Pivot-¿B with the associated subset and ponce for the base edge Pivot-¿A for the associated subset. If there are not points in a subset then stop the recursion.

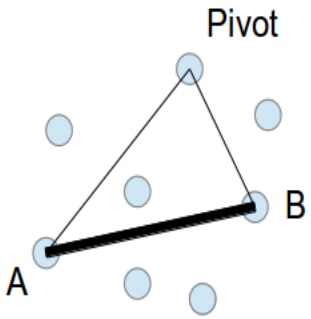

Figure 3: QuickHull Example

4 Algorithm Implementation

4.1 Brute Force

Our implementation for the brute force algorithm is shown in figure 4 below. There weren't any real issues getting it to work and it's fairly self explainitory.

```
public static LinkedList<Point2D> bruteForceConvexHull(Point2D points[]) {
    LinkedList<Point2D> convexHull = new LinkedList<Point2D>();

    double curr_x_prod, prev_x_prod;
    boolean on_hull;
    for(Point2D a: points) {
        for(Point2D b: points) {
            if(a != b) {
            prev_x_prod = Math.PI;
            on_hull = true;
            for(Point2D c: points) {
                if(c != a && c != b) {
                    curr_x_prod = ((a.getX() - b.getX())*(b.getY() - c.getY()) - (b.getX() - c.getX())*(a.getY() - b.getY()));
                    if(curr_x_prod > 0) {
                        curr_x_prod = 1;
                    }
                    else if(curr_x_prod < 0) {
                        curr_x_prod = -1;
                    }
                    else {
                        curr_x_prod = 0;
                    }
                    if (curr_x_prod == prev_x_prod || prev_x_prod == Math.PI || curr_x_prod == 0) {
                        prev_x_prod = curr_x_prod;
                    }
                    else {
                        on_hull = false;
                        break;
                    }
                }
            }
            if(on_hull) {
                if(!convexHull.contains(a)) {
                    convexHull.add(a);
                }
                if(!convexHull.contains(b)) {
                    convexHull.add(b);
                }
            }
            }
        }
    }

    return convexHull;
}
```

Figure 4: Brute Force Convex Hull

4.2 QuickHull

The implementation of QuickHull on the other hand ended up being very time consuming, mainly due to the difficulties inherint in debugging recusive code.

```
private Point2D [] max_min(Point2D generated_points[]) {
    Point2D[] min_max = new Point2D[2];
    Point2D min_x_pt = generated_points[0];
    Point2D max_x_pt = generated_points[0];
    for(Point2D pt: generated_points) {
        if(pt.getX() > max_x_pt.getX()) {
            max_x_pt = pt;
        }
        else if(pt.getX() < min_x_pt.getX()) {
            min_x_pt = pt;
        }
    }
    min_max[0] = min_x_pt;
    min_max[1] = max_x_pt;
    return min_max;
}
```

Figure 5: Max Min Function

The maxmin() function above is used to find the initial base edge composed of the furthest left and right points. Basically just a loop through all points keeping track of the current highest and lowest points.

```
private double get_determinant(Point2D a,Point2D b,Point2D c) {
    return a.getX()*b.getY() + c.getX()*a.getY() + b.getX()*c.getY() - c.getX()*b.getY() - b.getX()*a.getY() - a.getX()*c.getY();
}
```

Figure 6: Calculate Determinant

Similar to the brute force case a determinant is used to figure out which side of the line a point is on. The method above is called each time a new pivot point is found in order to find points that lay either to the left or the right of the resulting triangle.

```
private Point2D get_pivot_point(Point2D[] edge, LinkedList<Point2D> subset) {
    double max_distance = 0, curr_distance;
    Point2D pivot_point = new Point2D(Math.PI, Math.PI);
    for(Point2D pt: subset) {
        curr_distance = Math.abs(get_determinant(edge[0], edge[1], pt));
        if((curr_distance > max_distance || curr_distance == 0) && !q_hull.contains(pt)) {
            max_distance = curr_distance;
            pivot_point = pt;
        }
    }
    return pivot_point;
}
```

Figure 7: Get The Pivot Point

Given a subset of points we need to find the pivot (furthest point from the base edge). This ends up being pretty straight forward since we already have a method to calculate the determinant, this can be repurposed in order to get the distance along a normal vector from the base edge to each point in the current set.

```
private LinkedList<Point2D> get_subset(Point2D[] edge, LinkedList<Point2D> point_set, int side) {
    LinkedList<Point2D> subset = new LinkedList<Point2D>();
    double curr_side;
    for(Point2D pt: point_set) {
        curr_side = get_determinant(edge[0], edge[1], pt);
        if(pt != edge[0] && pt != edge[1]) {
            if(side > 0 && curr_side > 0) {
                subset.add(pt);
            }
            if(side < 0 && curr_side < 0) {
                subset.add(pt);
            }
            if(Math.abs(curr_side) == 0) {
                subset.add(pt);
            }
        }
    }
    return subset;
}
```

Figure 8: Get The Resulting Subset

One of the keys to getting QuickHull to work was figuring out which of the points in a given subset should be used to create the next subsets for the recursive call. This is again accomplished by looping through the remaining points and calling the determinanat method on each point, points to the right of a line on the upper right side of the pivot will have a positive determinant, points on the upper left side a negative determinant. For the lower hull these values are swithced.

```
private void dome(Point2D[] edge, LinkedList<Point2D> point_set, boolean upper) {
    Point2D pivot;
    Point2D[] right_edge = new Point2D[2];
    Point2D[] left_edge = new Point2D[2];
    LinkedList<Point2D> right_subset;
    LinkedList<Point2D> left_subset;

    if(point_set.size() > 0) {
        pivot = get_pivot_point(edge, point_set);
        if(pivot.getX() != Math.PI && pivot.getY() != Math.PI) {
            q_hull.add(pivot);
            if( edge[0].getX() > edge[1].getX()) {
                right_edge[0] = pivot;
                right_edge[1] = edge[0];
                left_edge[0] = pivot;
                left_edge[1] = edge[1];
            }
            else {
                right_edge[0] = pivot;
                right_edge[1] = edge[1];
                left_edge[0] = pivot;
                left_edge[1] = edge[0];
            }
            if(upper) {
                right_subset = get_subset(right_edge, point_set, 1);
                left_subset = get_subset(left_edge, point_set, -1);
            }
            else {
                right_subset = get_subset(right_edge, point_set, -1);
                left_subset = get_subset(left_edge, point_set, 1);
            }
            if(right_subset.size() > 0) {
                dome(right_edge, right_subset, upper);
            }
            if(left_subset.size() > 0) {
                dome(left_edge, left_subset, upper);
            }
        }
    }
}
```

Figure 9: Recursive Dome Function

The heart of this implementation of QuickHull is the recursive dome method. It taked the current edge, set of points and a flag indicating if the current subset is int he upper or lower hull. Next we find the pivot point and figure out if the resulting edges are on the left or the right side of the calling edge. and finally if there are any points in the new resulting subsets we can either call the dome function again or stop the recursion.

```
public void quickHull(Point2D points[]) {
    LinkedList<Point2D> generated_points = new LinkedList<Point2D>(Arrays.asList(points));
    Point2D[] base_edge = max_min(points);
    LinkedList<Point2D> upper_hull = get_subset(base_edge, generated_points, 1);
    LinkedList<Point2D> lower_hull = get_subset(base_edge, generated_points, -1);
    q_hull = new LinkedList<Point2D>();

    q_hull.add(base_edge[0]);
    q_hull.add(base_edge[1]);

    dome(base_edge, upper_hull, true);
    dome(base_edge, lower_hull, false);
}
```

Figure 10: Setup And Call Dome

The final component of this QuickHull implementation is the above initialization function appropriately called quickhull. As mentioned earlier we need to first find the furthest left and right points then proceed to divide the whole set into upper and lower subsets and finally make the calls to the recursive dome function with the appropriate point sets and flag.

5 Analysis of Time Complexity

5.1 Brute Force

Our brute force implementation is composed of 3 loops, each of which iterates through every point in the input. The basic operation for our algorithm is calculating the determinant which occurs in the inner most loop.

$$C(n) = \sum_{i=1}^{n}\sum_{i=1}^{n-1}\sum_{i=1}^{n-2} 1$$
$$C(n) = (n-2)\cdot\sum_{i=1}^{n}\sum_{i=1}^{n-1} 1$$
$$C(n) = (n-2)(n-1)\cdot\sum_{i=1}^{n} 1$$
$$C(n) = (n-2)(n-1)n$$
$$C(n) = n^3 - 3n^2 + 2n$$

$$O(n^3 - 3n^2 + 2n)$$
$$\boldsymbol{O(n^3)}$$

It's important to note that the calculations above do not take into account cases where we're able to break out of the inner loop early when we find points on either side of the line that is being tested. This means that specific arrangement of points we get as input will play a role in how quickly the algorithm completes. The best case should be a situation in which the points are completly random, we would quickly find a point that is not on the same side as the others and exit the loop early. On the other hand the worst case situation should occur with ordered input where we don't find a point on the other side of the line until late in the loop.

5.2 QuickHull

QuickHull is a divide and conquer type algorithm so we will use the **general divide-and-conquer reoccurance** in conjunction with the Master Theorem to analyis it. *[Levitin pg.171]*

$$T(n) = aT(n/b) + f(n)$$

Where a is the number of subsets needing to be solved, b is is the size of each subset, and $f(n)$ is the amout of time it takes to divide n into n/bi subsets.

Best Case (Using Master Theorem)

$a = b = 2$	[partition into two even subsets]
$f(n) = n$	[iterate through every point to find pivot]
$f(n) \epsilon \Theta(n^1)$	[so in our case $d = 1$]
$\boldsymbol{T(n) = 2T(n/2) + n}$	
$\boldsymbol{\Theta(n \log(n))}$	[since $a = b^1$]

Worst Case (Backward Substitution)

$T(n) = T(n-1) + c \cdot n$ [all points on one side of pivot]

$T(n) = T(n-2) + c[(n-1) + n]$

$T(n) = T(n-3) + c[(n-3) + (n-2) + (n-1) + n]$

$T(n) = T(n-4) + c[(n-4) + (n-3) + (n-2) + (n-1) + n]$

$T(n) = T(n-i) + c\sum_{j=0}^{i}(n-j)$

$T(n) = T(n-i) + c(n\sum_{j=0}^{i}1 - \sum_{j=0}^{i}j)$ [sum rules]

$T(n) = T(n-i) + c(n(\sum_{j=1}^{i}1 - 1) - \sum_{j=1}^{i}j)$ [sum rules]

$T(n) = T(n-i) + c(n(i-1) - \frac{i(i-1)}{2})$ [known sums]

$T(n) = T(n-n) + c(n(n-1) - \frac{n(n-1)}{2})$ [substituting $i = n$, $T(0) = 0$]

$T(n) = c(n^2 - n - \frac{n^2 - n)}{2})$

$T(n) = c(\frac{n^2}{2} - \frac{n}{2})$ [partial fraction expansion]

$O(c(\frac{n^2}{2} - \frac{n}{2}))$

$\boldsymbol{O(n^2)}$ [ignore lower order terms/constants]

5.3 Expected Outcome

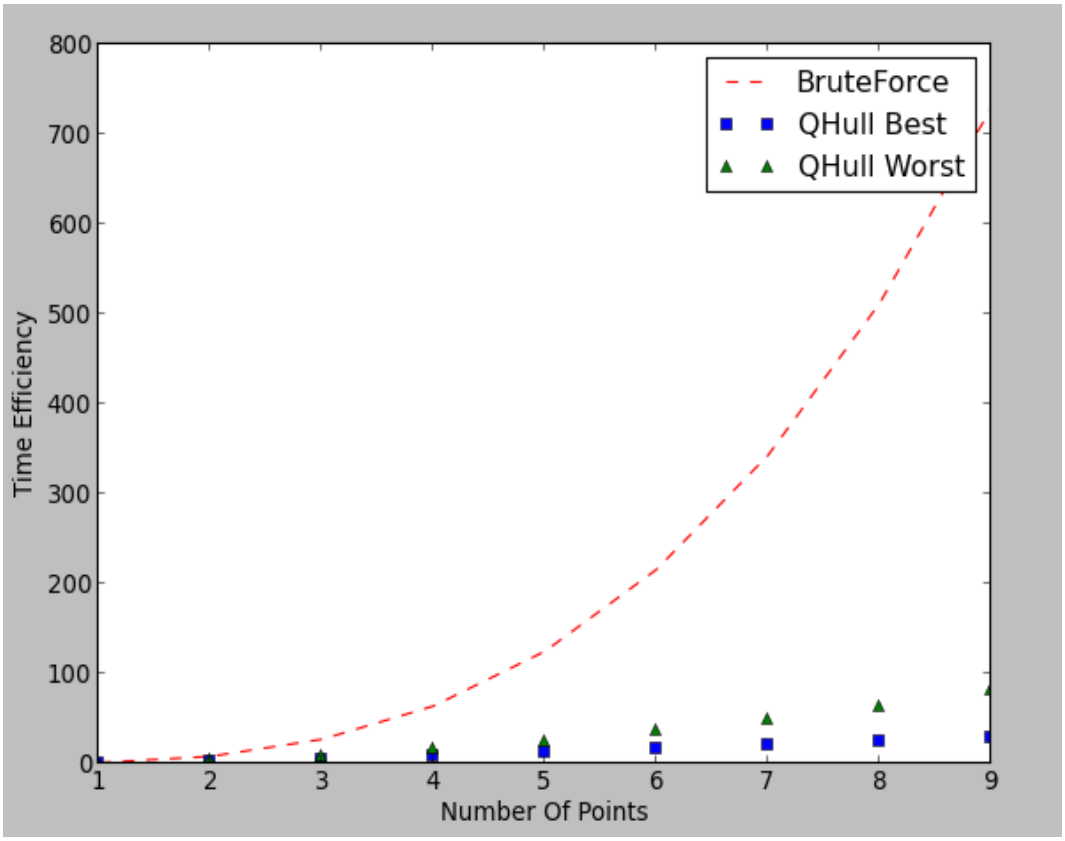

Figure 11: Time Efficiency Analysis

6 Automated Testing

6.1 Algorithm Correctness

When deciding on a strategy to test the correctness of our algorithm we had a few key criteria. We wanted to avoid repeating the steps used in our algorithms, we wanted to be able to use double precision so that we could test for correctness with circular data, and we wanted to use some well tested java libraries. These criteria brought us to the Polygon.contains method in the java.awt library, but the contains method only works on integers, and this would disable our ability to correctly test convex hulls that are in the shape of a circle. We then discovered the Path2D library which supports double precision and has a contains method. The challenge with using the Path2D library comes in building the convex hull. Building the hull requires adding points in the correct order so that each point you add draws the correct line as it would look in the given convex hull. In order to solve this problem we had to sort the convex hull in a circular way so that the path could be drawn clockwise around the convex hull.

```
LinkedList<Point2D> subList = new LinkedList<Point2D>();
int currentPosition = 0;

//top of the array
circularArray[currentPosition] = left;
currentPosition++;
for(Point2D current: cHull) {
    if(current.getX() > left.getX() && current.getX() < top.getX()
            && current.getY() > left.getY() && current.getY() < top.getY())
        subList.add(current);
    else if(current.getX() > top.getX() && current.getX() < right.getX()
            && current.getY() < top.getY() && current.getY() > right.getY())
        subList.add(current);
}
if(left != top && right != top)
    subList.add(top);

while(!subList.isEmpty()) {
    Point2D current = findLeftmostPoint(subList);
    subList.remove(current);
    circularArray[currentPosition] = current;
    currentPosition++;
}

circularArray[currentPosition] = right;
currentPosition++;
```

Figure 12: Circular Case

The code above shows the algorithm for building an array that can be used to draw the convex hull clockwise. First we add the left most point of the convex hull in the first position of the circular array. We start with the left, right, top, and bottom points of the convex hull available, and we use those to add the top left and top right portions of the convex hull to a linked list. We then find the left most element of the list and place it into the next available position of the circular array while removing it from the list. We repeat that step until the list is empty, and add the right point to the next available position. A similar strategy is used to add the bottom portion of the convex hull to the circular array, but the loop portion adds the right most point to the array to continue adding points in the clockwise direction. The code below shows the construction of the convex hull using the Path2D cHull. The path starts with the left most point contained in the first position of the circular array. The rest of the list is iterated through and lines are drawn to each consecutive point. A line is then drawn from the final point in the array to the first point added using the closePath method. With the convex hull drawn we are able to iterate through all the points and make sure each point is contained in the convex hull.

```
Point2D [] circularArray = getCircularArray(convexHullArray);

cHull.moveTo(circularArray[0].getX(), circularArray[0].getY());
for(int i = 1; i < circularArray.length; i++) {
    cHull.lineTo(circularArray[i].getX(), circularArray[i].getY());
}
cHull.closePath();


for(Point2D current: points) {
    totalCorrectness = totalCorrectness && cHull.contains(current.getX(), current.getY());
    System.out.println(cHull.contains(current.getX(), current.getY())
            + " " + current.getX() + " " + current.getY());
}
```

Figure 13: Circular Correctness Test

6.2 Data Generation

In order to accurately assess the efficiency of our algorithms we came up with two different ways to generate test data: circular points and random points. When given circular points the efficiency of quick hull should decrease significantly, because each time it draws a triangle and searches for points inside the triangle it will find nothing, so the convex hull would be the entire set of points. The random data generated would give closer to average case efficiency, because usually provided points would be closer to random distribution. In order to generate a circular set of points the function below received a count of how many points need to be generated. At the start of the loop degrees is set to 0, and is incremented by 360 divided by the total number of points so that the data will be distributed well around the circle. Each point is given an x value of the radius multiplied by the cos of the current value of degrees, and the y value is generated using the sin function. The random data is generated point by point and uses the Math.random function.

7 Results

To confirm that the actual running time of the algorithms matched with the efficiency class of the algorithms we ran our code with varying sizes of input. We used our data generation functions to create random input and circle input for varying values of n, and we used the same data for both the brute force and the quick hull algorithms. We looped through the quick hull and brute force algorithms enough times to see a clear difference between the two algorithms.

Testing the speed of the algorithm proved to be a challenge due to the extreme inefficiency of the brute force approach with large amounts of data. I attempted to run the brute force algorithm with 10,000 elements and waited 40 minutes before giving up. This led us to test with sizes 10, 100, and 1000. The table below shows the data we collected when running the tests.

Algorithm	Data Type	Number of Elements	Number of times run	Total Time taken(milliseconds)	Average Time taken(milliseconds)
Brute Force	Circular	10	2000	25	0
Brute Force	Random	10	2000	10	0
Quick Hull	Circular	10	2000	33	0
Quick Hull	Random	10	2000	8	0
Brute Force	Circular	100	200	262	1
Brute Force	Random	100	200	194	0
Quick Hull	Circular	100	200	42	0
Quick Hull	Random	100	200	5	0
Brute Force	Circular	1000	20	147371	7368
Brute Force	Random	1000	20	7368	64
Quick Hull	Circular	1000	20	5	0
Quick Hull	Random	1000	20	4	0
Quick Hull	Circular	10000	1	17	17
Quick Hull	Random	10000	1	4	4
Quick Hull	Circular	100000	1	79	79
Quick Hull	Random	100000	1	23	23

Figure 14: Results

Looking at quick hull and comparing it with brute force with 10 elements shows very similar running speeds, but as soon as 100 elements are tested the difference becomes

apparent. The rate of growth difference seems to be massive. With 100 elements of circular data brute force took 6 times longer to run than quick hull, and at 1000 elements of circular data brute force took 29,474 times longer to run than quick hull. While brute force took longer than 40 minutes to run before we quit the process on 10,000 elements quick hull took only 17 milliseconds.

We expected the data generated in circular form to perform in n squared time for quick hull because it is the worst case, and in n cubed time for brute force, so we assumed that the difference in running time would be a factor of n. With 100 elements the factor was only 6, and with 1,000 elements the factor was 29,474, and these factors are very different than the factors of 100 and 1,000 that we expected. This showed us that the performance of an algorithm with a specific efficiency class can vary drastically due to implementation details, and although constant factors are ignored in efficiency classes they make a big difference in practice.