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

2 ConvexHull Overview

3 Algorithm Explaination

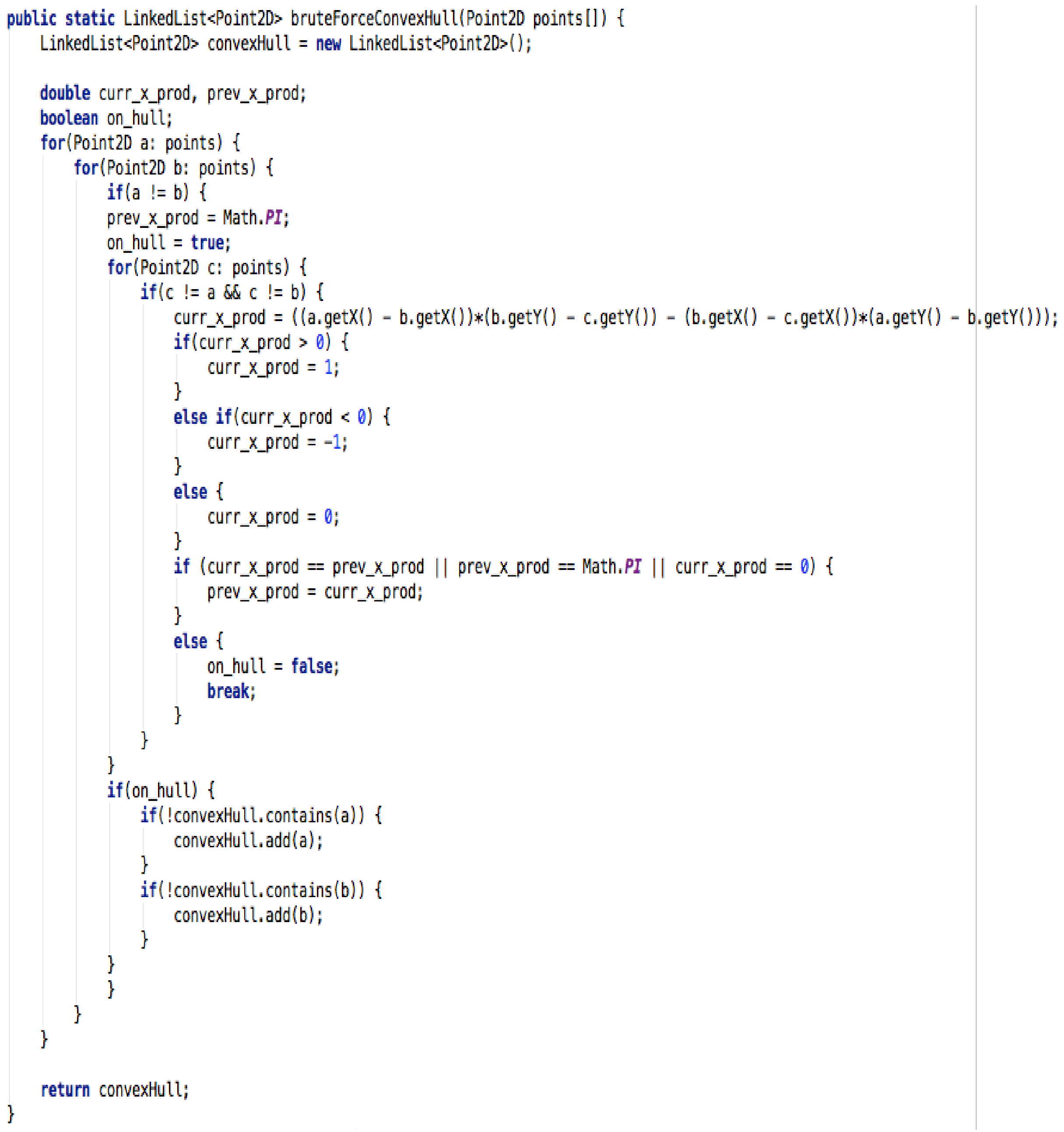
3.1 Brute Force

3.2 QuickHull

4 Algorithm Implementation

4.1 Brute Force

As the name implies the brute force solution to the convexhule problem is very straight forward, as shown below in Figure 1.



```java
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 1: Brute Force Convex Hull

4.2 QuickHull

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.

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

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

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]

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

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

5.3 Expected Outcome

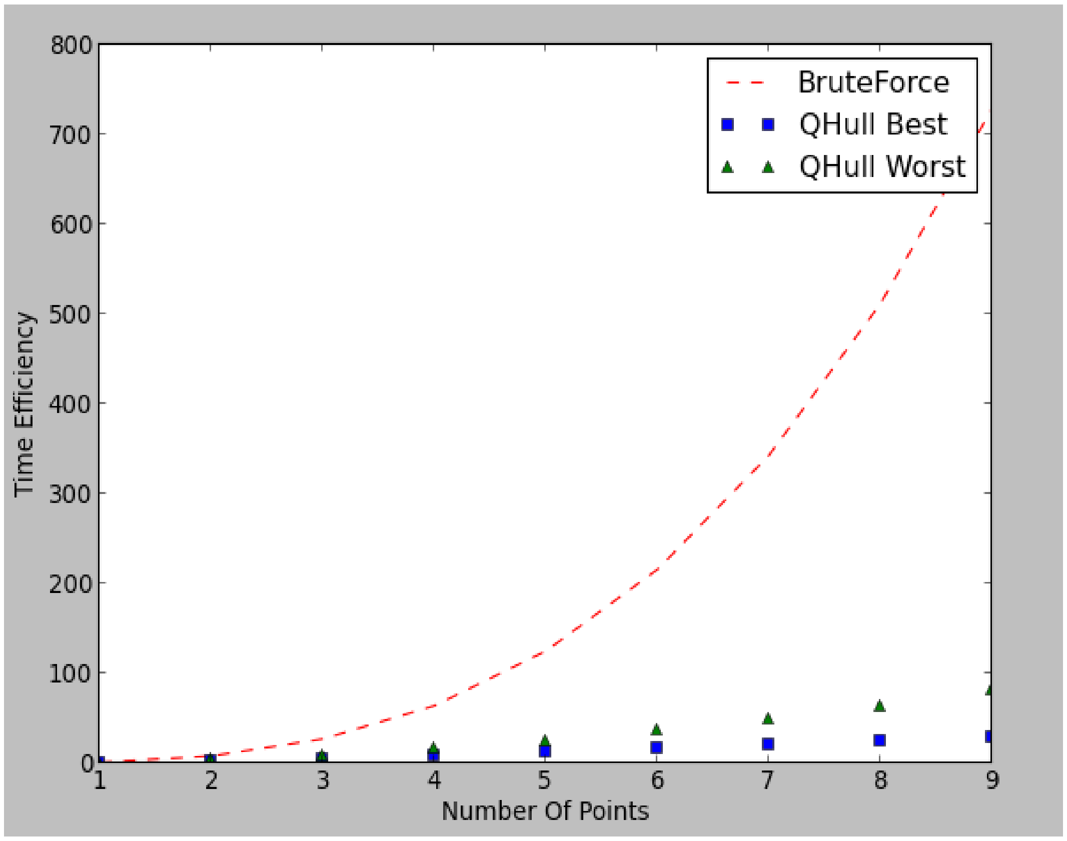
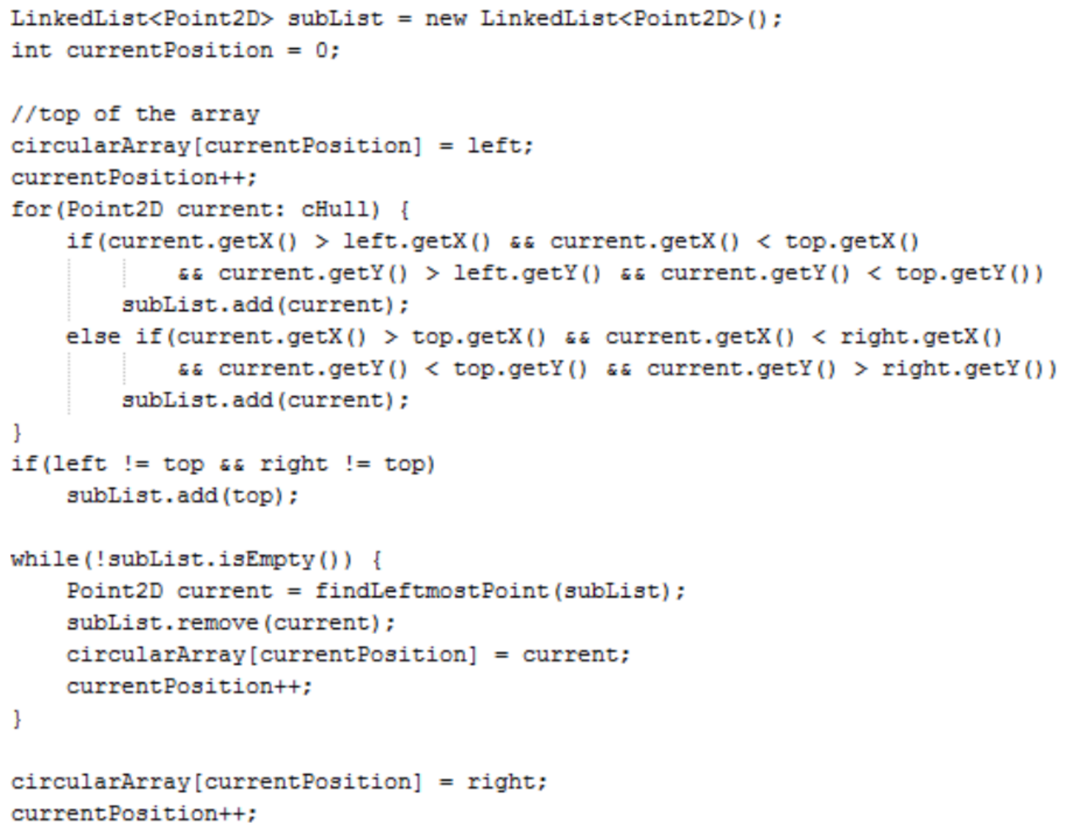


Figure 2: Time Efficiency Analysis

6 Automated Testing

6.1 Algorithm Correctness

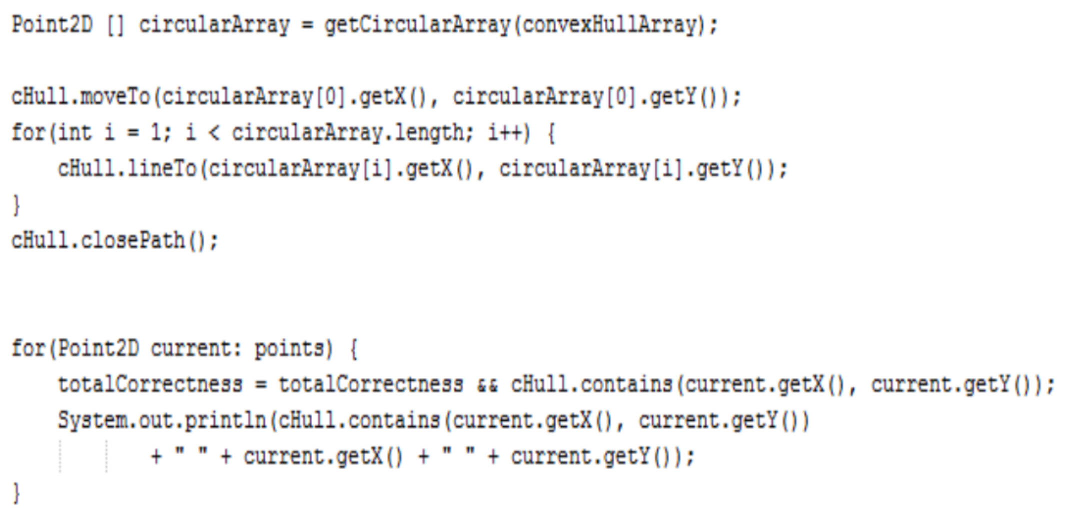
When deciding on a strategy to test the correctness of our algorithm we had a few key criteria. We wanted to avoid repeating the steps used in our algorithms, we wanted to be able to use double precision so that we could test for correctness with circular data, and we wanted to use some well tested java 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 3: 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 4: 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

8 Conclusion