Convex Hull Report

Emily Elzinga

September 2022

1 Pseudocode

1. The set of n points is divided into two subsets, L containing the leftmost $\frac{n}{2}$ points and R containing the rightmost $\frac{n}{2}$ points.

2. The convex hulls of the subsets L and R are computed recursively.

   (a) To merge the left hull CH(L) and the right hull CH(R), it is necessary to find the two edges known as the upper and lower common tangents (shown in red below). A common tangent of two simple convex polygons is a line segment in the exterior of both polygons intersecting each polygon at a single vertex. If continued infinitely in either direction, the common tangent would not intersect the interior of either polygon.

   (b) The upper common tangent can be found by scanning around the left hull in a counter-clockwise direction and around the right hull in a clockwise direction. Start with the rightmost point of the left hull and the leftmost point of the right hull. While the edge is not upper tangent to both left and right. While the edge is not upper tangent to the left, move counter-clockwise to the next point on the left hull. Hint: We want to move to the next point(s) on the left hull as long as the slope decreases.While the edge is not upper tangent to the right, move clockwise to the next point on the right hull.

Taken together the convex hull problem can be solved with the following function. This is the driver for the algorithm. It takes the series of points and divides them up into 2 subproblems, of size n/2. According to the master theorem, a = 2, b=2, and d =2. Therefore the time complexity of the algorithm is n^2.

```
    def divide_and_conquer(self, points):
        convex_hull = []
        if len(points) >= 2:
            left_hull = self.divide_and_conquer(points[:int(len(points) / 2)])
            right_hull = self.divide_and_conquer(points[int(len(points) /
              ↪ 2):])
        else:
            return points               #base case of one point
        maximum = self.largest(left_hull)  # rightmost point of left
        minimum = self.smallest(right_hull)
        ul, ur = self.find_upper(left_hull, right_hull, minimum, maximum)

        ll, lr = self.find_lower(left_hull, right_hull, minimum, maximum)

        convex_hull = self.make_new_hull(ul, ll, left_hull,convex_hull)
        return self.make_new_hull(lr,ur, right_hull,convex_hull)
```

This method makes a side of a new hull. It starts at the lower left and looks for the upper left node to make the left side of the new hull. The right side is then passed in and it starts at the lower right node and looks for the upper right node. It is linear in time complexity.

```
    def make_new_hull(self, end, start, side, new_hull):
        encountered = False
        i = side.index(start)
        while encountered == False and i <= len(side):
            if side[i] == end: encountered = True
            new_hull.append(side[i])
            i = i+1
            if encountered == False and i >= len(side):
                i = 0
                #if the end of the hull array has been reached and the ul/lr
                  ↪ point has not been found, restart
        encountered = False

        return new_hull
```

Finds the smallest element in the hull array. Linear in time complexity.

```
def largest(self, arr):
    # Initialize maximum element
    max = arr[0]
    max_index = 0
    # Traverse array elements from second
    # and compare every element with
    # current max
    for i in range(1, len(arr)):
        if QPointF.x(arr[i]) > QPointF.x(max):
            max = arr[i]
            max_index = i
    return max_index
```

Finds the point with the smallest x-value in the hull array. Linear in time complexity.

```
def smallest(self, arr):
    # Initialize min element
    min = arr[0]
    min_index = 0
    # Traverse array elements from second
    # and compare every element with
    # current max
    for i in range(1, len(arr)):
        if QPointF.x(arr[i]) < QPointF.x(min):
            min = arr[i]
            min_index = i
    return min_index
```

This method attempts to find the upper tangent points. It starts with the points with the minimum and maximum x-values and attempts to iterate counterclockwise over the left hull and clockwise over the right. This works much of the time, but there are always edge cases that wreak havoc on further calculations. The time complexity of this method is n^2.

```
def find_upper(self, left_hull, right_hull, minimum, maximum):
    upper_left = left_hull[maximum]               #maximum x value of left
      ↪ hull
    upper_right = right_hull[minimum]              #minimum x value of
      ↪ right hull
    # leftmost point of right #slope of current two points
    slope0 = self.find_slope(upper_right, upper_left)

    i = maximum-1

    while i != maximum:
        if i == 0 and len(left_hull) != 1:
            i = len(left_hull)-1
            # once past rightmost point, go to back of the array and
        if(len(left_hull)==1): break
        point = left_hull[i]
        i = i - 1  # go counterclockwise


        slope1 = self.find_slope(upper_right, point)
        if slope1 <= slope0:
            for point0 in right_hull:
                slope12 = self.find_slope(upper_left, point0)
                if slope12 < slope0: break
                slope0 = slope12
                upper_right = point0
        upper_left = point

    return upper_left, upper_right
```

This method attempts to find the lower tangent points of the polygon. As with $find_upper$, it starts with the points with the minimum and maximum x-values and attempts to iterate clockwise over the left hull and counterclockwise over the right hull. The complexity of this method is n^2.

```
    def find_lower(self, left_hull, right_hull,minimum, maximum):

        lower_left = left_hull[maximum]                #maximum x_value of
          ↪ left hull
        lower_right = right_hull[minimum]              #minimum x_value of
          ↪ right hull

        slope0 = self.find_slope(lower_right, lower_left)
        #slope of current two points
        #
        #
        j = maximum - 1

        # for index in (range(0, len(left_hull))):
        while j >= 0:

            point = left_hull[j]
            slope1 = self.find_slope(lower_right, point)
            j = j - 1

            if slope1 >= slope0 or left_hull[maximum] == point:
                i = minimum - 1
                while i != minimum:
                   if i == minimum -1 and len(right_hull) != 1: i =
                     ↪ len(right_hull)
                   elif len(right_hull) == 1: break
                   #once past leftmost point, go to back of the array and
                   i = i-1                          #go counterclockwise


                   point0 = right_hull[i]
                   slope12 = self.find_slope(lower_left, point0)
                   if slope12 > slope0: break
                   slope0 = slope12
                   lower_right = point0

                lower_left = point

        return lower_left, lower_right
```

2 Checkist

40 Correct functioning code to solve the Convex Hull problem using the divide and conquer

scheme discussed above. Include your documented source code. This isn't quite completed. In the way that it is currently implemented, there are edge cases that result in incorrect tangent nodes and wreak havoc on future calculations. I am not currently sure how to fix it so that it does work in every case.

15 Include the full, unambiguous pseudo-code for your divide-and-conquer algorithm for finding the convex hull of a set of points Q. Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your pseudo-code. The pseudocode is shown above. I wasn't sure how to format the psuedocode, since its syntax is vastly similar to Python.

1. The time complexity of this algorithm is n^2.

2. Also, I used GeeksforGeeks' implementations of the Quicksort[**?**] and max[**?**] and min value sorting algorithms.

3 Screenshots

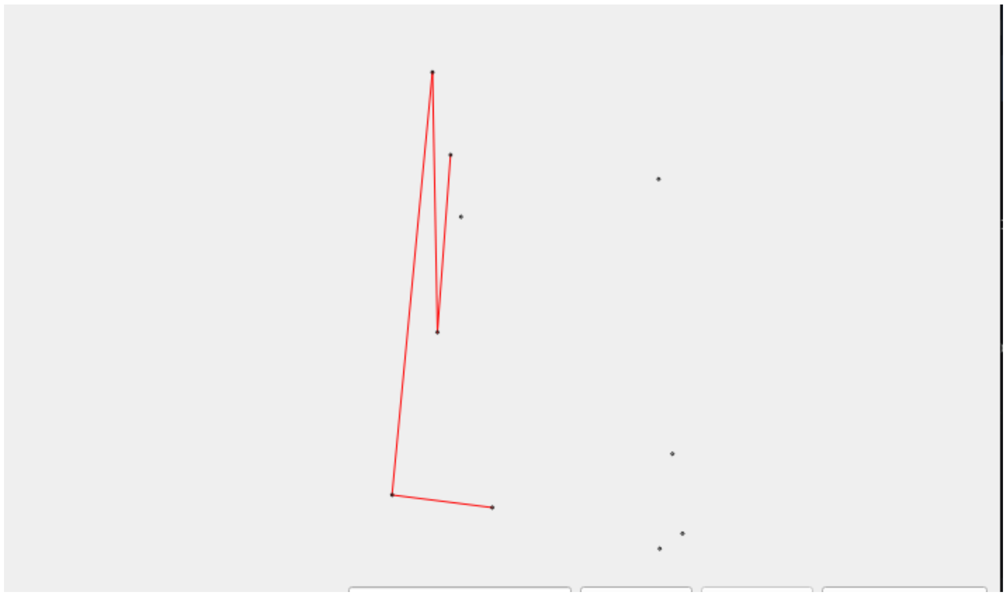

Figure 1 – The result with 10 points

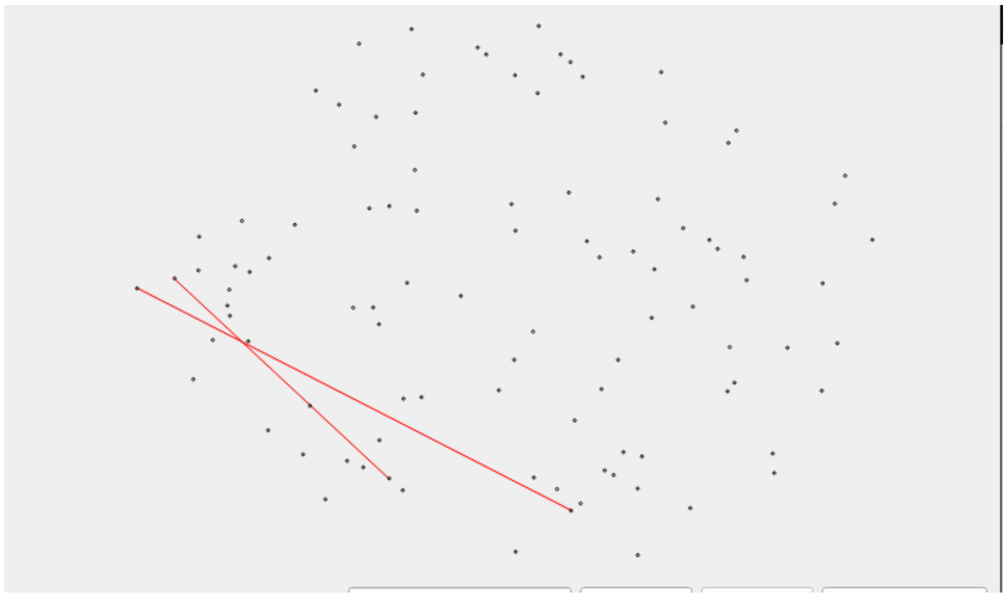
Figure 2 – The result with 100 points