American Journal of Engineering Research (AJER) 2016
American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-5, Issue-11, pp-70-75
www.ajer.org
Research Paper Open Access

A Mean Point Based Convex Hull Computation Algorithm

Digvijay Singh[1], Hemang Sarkar[2], L N Das[3]
[1,2](Department of Applied Mathematics, Delhi Technological University, India)
[3](Associate Professor, Department of Applied Mathematics, Delhi Technological University, India)

ABSTRACT: *The optimal solution of a Linear Programming problem (LPP) is a basic feasible solution and all basic feasible solutions are extreme or boundary points of a convex region formed by the constraint functions of the LPP. In fact, the feasible solution space is not always a convex set so the verification of extreme points for optimality is quite difficult. In order to cover the non-convex feasible points within a convex set, a convex hull is imagined so that the extreme or boundary points may be checked for evaluation of the optimum solution in the decision-making process. In this article a computer assisted convex hull computation algorithm using the Mean Point and Python code verified results of the designed algorithm are discussed.*

Keywords: *Boundary points, Convex set, Convex hull, Extreme points, Mean Point, LPP*

I. INTRODUCTION

In Euclidean space, a convex set is the 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. The convex hull or convex envelope of a set *X* of points in the Euclidean plane or Euclidean space is the smallest convex set that contains *X*. The domains and ranges of the functions for many linear programming problems' constraints are hyper-planes which are convex sets. The solution processes of optimization problems are computed by improving the finite set of feasible points by means of a suitable iterative algorithm. The set of feasible points in many situations are not defined in the convex set forms. Therefore, a minimum convex set containing the feasible solution set in the form of a convex hull is computed first to decide the possible convex polyhedron that contain the basic feasible solution (BFS) as its vertices.

In computational geometry, several algorithms are proposed for the computation of the convex hull of a finite set of points with various computational complexities. The convex hull computation is the framing of a convex shape that can represent the required convex feasible region for the constrained optimization problem. The time complexity of the corresponding algorithm is usually estimated in terms of n - the number of input points of the finite set and/or h - the number of points on the boundary of the convex-hull. If the time complexity depends on h, then the algorithm is output sensitive. There are several algorithms available in the convex- hull computation algorithm literature.

The notable algorithms are Gift wrapping planar algorithm of Jarvis [1], Graham Scan algorithm [2], Chan's algorithm [3], The ultimate planar convex hull algorithm of Kirkpatrick & Seidel [4] and Akl-Toussaint heuristic algorithm [5]. In section II, we present the pseudocode of the algorithm. Then in section III, we give a step by step description of the algorithm. Section IV presents the analysis of the computational complexity of the algorithm followed by section V where the outputs of the Python code for the algorithm are presented.

II. PSEUDOCODE OF THE PROPOSED ALGORITHM

```
check_counter_clockwise(prev, nxt, curr)
1-  value = (nxt.x - prev.x)*(curr.y - prev.y) - (nxt.y - prev.y)*(curr.x - prev.x)
2-  if value > 0:
```

```
3-      return False
4-  else:
5-      return True
```

convexhull(list A):

1- mean_x = mean x-coordinate in list A
2- mean_y = mean y-coordinate in list A
3- M = [mean_x, mean_y]
4- max_point = point in list A with maximum x coordinate (take the first instance in case of multiple instances)
5- min_point = point in list A with minimum x coordinate
6- Initialise empty lists Q_1 = [ ], Q_2 = [ ], Q_3 = [ ], Q_4 = [ ]
7- for i = 1 to n:
8- If i doesn't correspond to max_point or min_point:
9- d = distance between A[i] and M
10- a = positive angle in degrees that the line joining A[i] with M makes with the line joining M with max_point in the counterclockwise direction
11- Append [A[i].x, A[i].y, d, a] to Q_1, Q_2, Q_3 or Q_4 according to the quadrant of A[i] with respect to M
12- Sort Q_1, Q_2, Q_3 and Q_4 according to distance parameter stored at each index
13- q_1 = length(Q_1), q_2 = length(Q_2), q_3 = length(Q_3) and q_4 = length(Q_4)
14- Initialise circular linked as convex_hull and insert max_point and min_point in it
15- while max(q_1, q_2, q_3, q_4) > 0:
16- Take out the last element from each of Q_1, Q_2, Q_3 and Q_4 (if it exists) and insert it in sorted order with respect to angle parameter in convex_hull
17- prev = convex_hull.head, curr = prev.next, nxt = curr.next
18- while curr is not convex_hull.head:
19- if check_counter_clockwise(prev, nxt, curr) is false:
20- curr = nxt
21- nxt = nxt.next
22- prev.next = curr
23- else:
24- prev = prev.next
25- curr = curr.next
26- nxt = nxt.next
27- Print convex_hull to get the points on the boundary of the convex hull in counterclockwise direction

III. DESCRIPTION OF THE ALGORITHM

(1) The input is a list of n pairs $[x, y]$ which denotes the set of input points on the X-Y plane for which we have to find the convex hull.

(2) Find the mean x- coordinate and the mean y- coordinate. Label this point as M[mean_x, mean_y]. This is the mean point.

(3) Since the two points with the maximum x coordinate and the minimum x coordinate have to be a part of the convex hull, we find them both and store them in max_point and min_point respectively.

(4) Initialise four lists labeled as Q_1, Q_2, Q_3 and Q_4 which denote the four quadrants when we draw imaginary X-Y axes by setting the origin at the point M. The positive direction of this X axis is the extension of the line segment joining M to the point max_point.

(5) The points- max_point and min_point are not put in any of these lists. Put every other input point in exactly one of these four lists by comparing their x coordinates and y coordinates with that of point M.

(6) Each value in these four lists is of the form [x coordinate, y coordinate, distance, angle]. Here distance is

the distance of the point from M. The angle is found as the inverse tangent of the ratio of length of y component (of line joining point to the M) with the x component (of the same line joining point to M) in degrees, and the points are considered from all four quadrants Q_1, Q_2, Q_3 and Q_4. We then add 360º to the values that are negative in order to store all angles as positive values.

(7) Sort the values in the four lists individually according to the distance parameter.

(8) Initialize a circular linked list called convex_hull and insert in it the two points: max_point and min_point. The head of the linked list points to the node corresponding to max_point. The node for max_point points to the node containing min_point which, in turn points to the head. This linked list at the end of the algorithm will have all the vertices of the convex hull in anti-clockwise direction starting at max_point. The nodes in this linked list will also be of the form [x coordinate, y coordinate, distance, angle]. Suppose curr is a node then curr.next gives us the node that comes after curr in the linked list.

(9) Let q_1, q_2, q_3 and q_4 be the number of points in Q_1, Q_2, Q_3 and Q_4 respectively. Repeat the steps (10) to (12) while $max(q_1, q_2, q_3, q_4) > 0$.

(10) Remove the last element from each list (if it exists otherwise pass) and store it in a new list temp_list. The last element is the current element with the maximum distance from M in that list. Whenever we remove an element, we decrease the corresponding variable q_1, q_2, q_3 or q_4.

(11) The number of elements in temp_list will be at most 4. We now insert these points into convex_hull based upon the angle parameter. The convex_hull list must always be in sorted order with respect to angle of the nodes starting at the first node (max_point with angle 0°). We insert a node with angle B between nodes with angles A and C if A< B < C. Note that a node might have to be inserted after the last node and before the head node if its angle is greater than every value in the list.

(12) After the insertion of the new nodes, we test whether the points can be a part of the boundary of the convex hull or not. In convex_hull, we start from the head node (max_point) and take three consecutive nodes at a time. These three points are in the counterclockwise direction since we inserted them in sorted order. Let these points be $prev: (x_1, y_1)$, $curr: (x, y)$ and $nxt: (x_2, y_2)$ in this order. These may be a part of the convex hull if and only if

$$(1): (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) \geq 0$$

If this value comes out to be negative, we delete the node curr denoting the point (x, y) and set $prev = (x_1, y_1)$, $curr = (x_2, y_2)$ and $nxt = nxt.next$

Otherwise, $prev = (x_1, y_1)$, $curr = (x_2, y_2)$ and $nxt = nxt.next$

This step is terminated when we go around the list once and reach the head of convex_hull again.

(13) After the while loop in from steps (9) to (12) ends, print the values in convex_hull starting from the head to get the vertices of the convex hull boundary in counter clockwise direction. These points are on the boundary of the convex hull.

IV. DESCRIPTION OF COMPUTATIONAL COMPLEXITY OF ALGORITHM AND INFERENCE

If we consider basic mathematical operations to run in constant time, we can see that the execution time of steps (2) to (6) is $O(n)$. Using a sorting algorithm like the merge sort or the quick sort, step (7) runs in $O(nlogn)$ time. This is so because we can argue that on average, one quadrant as defined above will have $\left\lfloor \frac{n}{4} \right\rfloor$ number of points.

The while loop started in step (9) executes as long as $q_1 > 0$ or $q_2 > 0$ or $q_3 > 0$ or $q_4 > 0$. The average number of times the while loop will run is thus $\left\lfloor \frac{n}{4} \right\rfloor$.

In each iteration of the while loop, we extract at most one point from each of the four quadrants and insert these points at their appropriate positions into convex_hull. After this, we iterate over this circular linked list considering three consecutive points at a time. Assume that we are at the i^{th} iteration and let the length of the

circular linked list be $L(i)$.

(2): $L(1) = 2$

We insert 1, 2, 3 or 4 points into this linked list in sorted order. Let $a(i)$ be the number of points inserted into the linked list in the ith iteration.

We have to count the average (expected) number of comparisons that are done for each of these insertions. For the first insertion, the expected number of comparisons will be $\frac{L(i)+1}{2}$. For the second insertion, the expected number of comparisons will be $\frac{L(i)+2}{2}$. Similarly, we can find the expected number of comparisons for the other two points. Hence the average number of comparisons will be:

(3): $\frac{(L(i)+1)+f(a(i),2)(L(i)+2)+f(a(i),3)(L(i)+3)+f(a(i),4)(L(i)+4)}{2}$

where the function $f(x,y)$is defined as:

(4): $f(x,y) = 1\ if x \geq y$ otherwise $f(x,y) = 0$

Since we are looking for an upper-bound to the number of comparisons, we assume that we make all four insertions in each case. The length of the list now is $(L(i) + 4)$ and we also see that the number of comparisons on average is $(2L(i) + 5)$ from (3).

After the insertions are done, we go around the list once and delete some point(s) based on the inequality (1). This adds to $(L(i) - 1)$ more operations. The number of the operations in the while loop started in step (9) of the algorithm is:

(5): $2L(1) + 2L(2) + \ldots + 2L\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + 5\left\lfloor\frac{n}{4}\right\rfloor + L(1) + L(2) + .. + L\left(\left\lfloor\frac{n}{4}\right\rfloor\right) - \left(\left\lfloor\frac{n}{4}\right\rfloor\right)$

Whenever we have u points, in the worst case we can delete $(u - 3)$ of them when we iterate around the list taking three consecutive points at a time. The minimum number of points deleted here can be 0. The expected number of deletions will be thus $\frac{(u-3)}{2}$. We also defined h - as the number of points which form the boundary of the convex hull. Assuming uniform distribution, we can claim that after i iterations, we will have $\left\lfloor\frac{4hi}{n}\right\rfloor$ number of points in our convex_hull which are a part of the boundary. These can't be deleted when we make a transition from the iteration $i$to the iteration $(i + 1)$of the while loop.

The expected length of the list after the deletion will be

(6): $L(i+1) = (L(i) + 4) - \left(\frac{L(i)+4-\left\lfloor\frac{4hi}{n}\right\rfloor-3}{2}\right)$

(7): $2L(i+1) = L(i) + 7 + \left\lfloor\frac{4hi}{n}\right\rfloor$

Putting values of i from 1 to$\left\lfloor\frac{n}{4}\right\rfloor$ in (7) and adding the equations vertically, we get:

(8): $L(1) + L(2) + \ldots + L\left(\left\lfloor\frac{n}{4}\right\rfloor\right) = O(nh)$

Using (8) with (5), we can see that the average runtime of the algorithm will be $O(nlogn) + O(n) + O(nh) = O(nh)$. This algorithm is thus output sensitive. In cases where $h < log(n)$, the runtime will be $O(nlogn)$.

The worst case occurs when all of the input points lie on the boundary of the convex hull, because then no point is deleted as we go from the i^{th} iteration to the $(i+1)^{th}$ iteration as discussed above and then we can show that the runtime is achieved by setting $n = h$ to get $O(n^2)$.

V. VERIFICATION OF CONVEX HULL COMPUTATIONAL ALGORITHM IN PYTHON CODE

We have encoded the above described algorithm in Python version 3.4 and used "matplotlib" to plot the geometric shape of the computed circular linked list of the points that forms the boundary shape of the convex hull. The set of input points were generated by pseudo random number generator function in the random module of Python. Instance outputs for the algorithm at $n = 10,100$ and 1000 are shown in Fig.1, Fig.2 and Fig.3 respectively. The point shown in red colour is the Mean Point for the input set. The dashed vertical and horizontal lines drawn through this point denote the imaginary X and Y axes required to separate the input set into Q_1, Q_2, Q_3 and Q_4. The points on the boundary of the convex hull region (both vertices and non-vertices) are shown in green colour and the interior points are shown in blue colour.

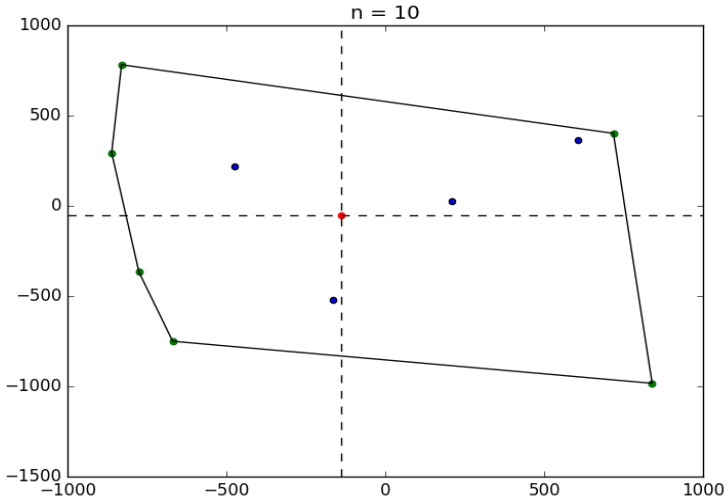

Fig. 1

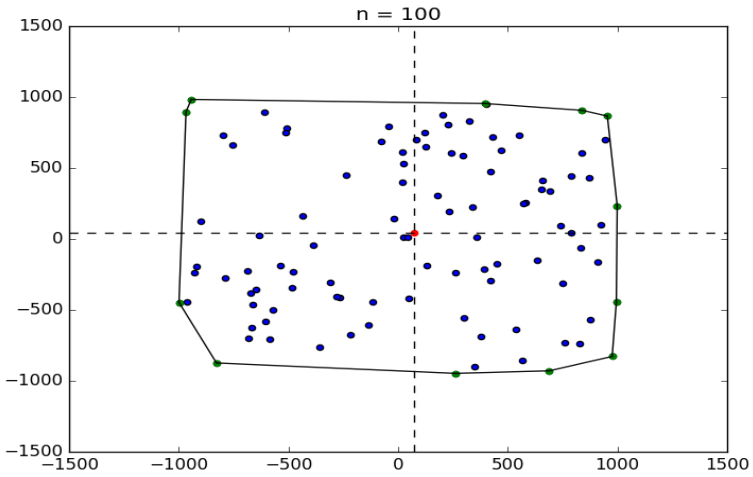

Fig. 2

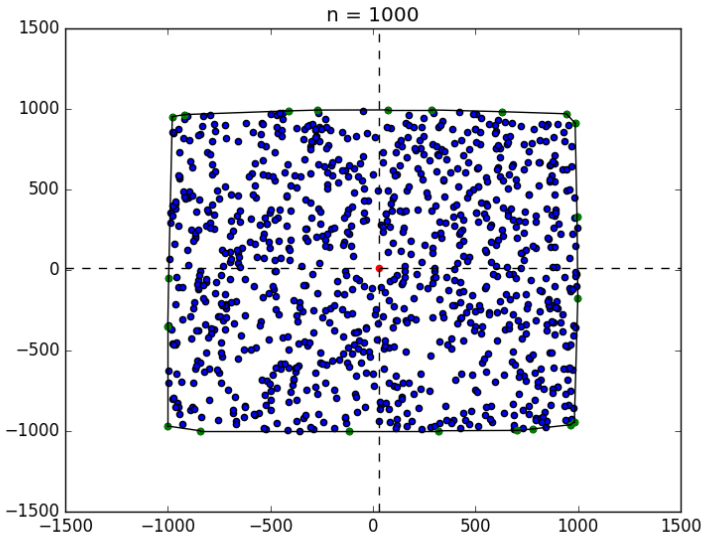

Fig. 3

VI. CONCLUSION

The discussed convex hull computation algorithm uses the mean point to find the polar angles and distances of all the input points and then with the help of a circular linked list implementation keeps track of all the points on the boundary of the convex hull. The average case time complexity of the algorithm is $O(nh)$. Thus, the runtime of the algorithm is the same as that of Jarvis [1].

REFERENCES

[1]. R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett., 2:18–21, 1973.*

[2]. Graham, R. L.: An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters1, 132-133 (1972)*

[3]. Chan, T.M. *Discrete Comput Geom (1996)16: 361. doi:10.1007/BF02712873*

[4]. Kirkpatrick, D. G., Seidel, R. (1986) The ultimate planar convex hull algorithm, *SIAM journal on Computing 15 (1), 287-299, doi:10.1137/0215021*

[5]. Selim G. Akl, & G. T, Toussaint, (1978) A fast convex hull algorithm, *Information Processing Letters, vol 7, pp 219-222.*