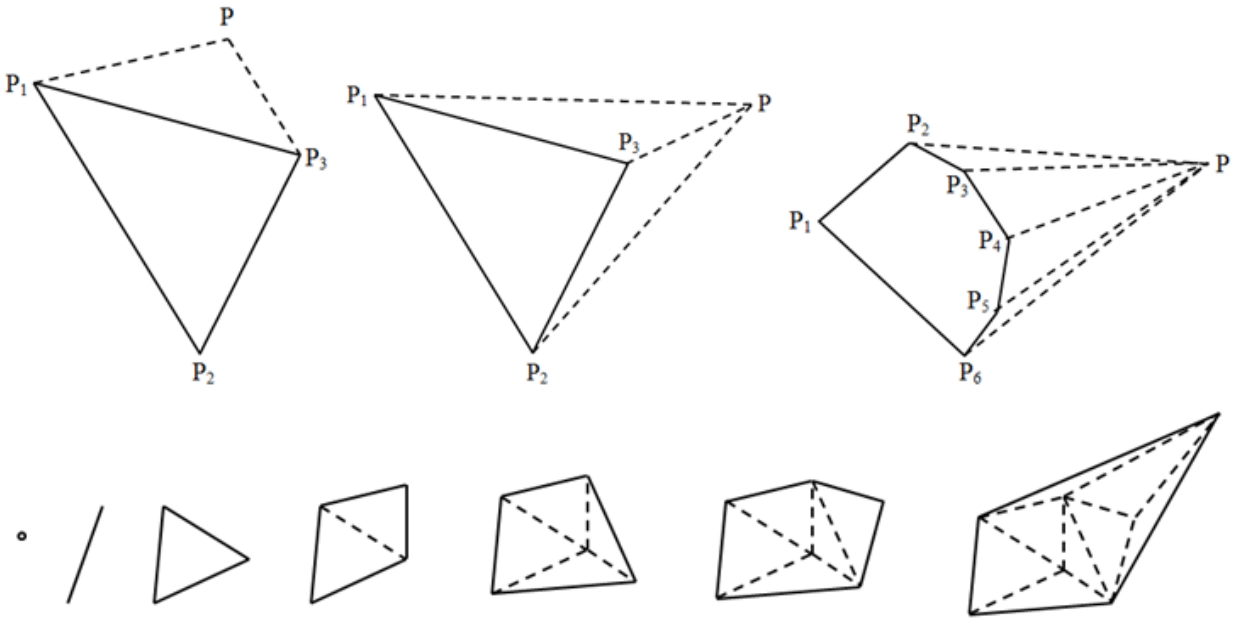
RESEARCH ARTICLE

An efficient planar incremental convex hull algorithm to find the edges of the boundary polygon of the convex hull of a set of points

K. R. Wijeweera* and S. R. Kodituwakku



Highlights

- A novel incremental convex hull algorithm is proposed with O (h^2) time complexity.
- The convex hull is maintained as a set of line segments.
- The proposed algorithm is faster than the existing algorithm when h < SQRT (n).

Ceylon Journal of Science 50(3) 2021: 261-268
DOI: http://doi.org/10.4038/cjs.v50i3.7907

RESEARCH ARTICLE

An efficient planar incremental convex hull algorithm to find the edges of the boundary polygon of the convex hull of a set of points

K. R. Wijeweera[1,2*] and S. R. Kodituwakku[1,3]

[1]Postgraduate Institute of Science, University of Peradeniya, Peradeniya, Sri Lanka.
[2]Department of Computer Science, Faculty of Science, University of Ruhuna, Matara, Sri Lanka.
[3]Department of Statistics and Computer Science, Faculty of Science, University of Peradeniya, Peradeniya, Sri Lanka.

Received: 05/07/2020; Accepted: 18/06/2021

Abstract: The definition of the convex hull of a set of points is the smallest convex set containing all the points. Many algorithms have been proposed with the worst case time complexity is equal to O (n log n). It has been proved that the lower bound of time complexity for construction of the convex hull is O (n log n). However, these algorithms are static and require all points at the start. Suppose that points reach sequentially one after one and the convex hull needs to be maintained at each insertion of a point. The convex hull should be constructed from the scratch to handle each point insertion if a static convex hull algorithm was used. This process consumes O (n log n) time and therefore it is inefficient. An incremental convex hull algorithm can maintain the convex hull at each insertion performing a trivial modification to the existing convex hull without constructing the convex hull from the scratch. The optimal time complexity for insertion of a point in existing incremental convex hull algorithms is O (n). A new incremental convex hull algorithm with O (h^2) time complexity is proposed in this paper. Note that *h* represents the number of vertices of the convex hull. A set of line segments is used to represent the convex hull. Some of the existing line segments may be deactivated instead of deleting upon the successive growth of the convex hull. Thus, the computational cost is reduced. The proposed algorithm is faster than the existing algorithms when h < SQRT (n) and the concept can be extended to three dimensions.

***Keywords*:** Convex Hull; algorithms; computational geometry; data structures; computational complexity.

INTRODUCTION

The convex hull is one of the most common structures found in the field of computational geometry. It is used in its own right and also as a tool to build other structures. The convex hull of a point set is defined as the smallest convex set containing all the points (O'Rourke, 1997).

The convex hull concept is widely used in computational geometry as well as in its applications. The applications dominate a wide variety of areas: image processing, statistics, analysis of spectrometry data, cartography, pattern recognition, file searching, crystallography, cluster analysis, collision detection, numerical integration, metallurgy, etc. Furthermore, there are some problems that can be brought down to the convex hull problem: half-space intersections, Voronoi diagrams, and Delaunay triangulations (Bayer, 1999).

Many algorithms have been proposed with the worst case time complexity is equal to O (n log n) (Akl and Toussaint, 1978; Bently and Shamos, 1978; Graham, 1972; Preparata, 1979; Preparata and Hong, 1977; Shamos, 1978a). It has been proved in several research papers (Preparata and Hong, 1977; Shamos, 1978a; Avis, 1979; Boas, 1980; Yao, 1981) that the lower bound of time complexity for convex hull construction is O (n log n). Two algorithms (Eddy, 1977; Jarvis 1973) have been proposed with O (nh) worst case time complexity. Note that *h* is the number of vertices on the convex hull.

Kirkpatrick and Seidel (Kirkpatrick and Seidel, 1986) proposed a two dimensional convex hull algorithm with O (n log h) worst case time complexity. Furthermore, they proved that O (n log h) is the theoretical lower bound of the problem.

Algorithms have also been proposed with linear expected time complexity (Bently and Shamos, 1978; Shamos, 1981b). Further, Akl, Toussaint, and Devroye (Akl and Toussaint, 1978; Devroye and Toussaint, 1981; Toussaint *et al*., 1978) have proved that a preprocessing step enable any of the algorithms mentioned above to run in O (n) expected time for particular distributions of input (Bayer, 1999). Practical running times have also been measured in (Bhattacharya and Toussaint, 1981).

All the algorithms discussed above are called static as they require all points at the start. Suppose that points reach sequentially one after one and the convex hull should be maintained at each point insertion. The convex hull should be constructed from the scratch if a point insertion is handled by a static convex hull algorithm. That means it needs O [(n + 1) log (n + 1)] = O (n log n) time to insert a point if there are *n* points already available (O'Rourke, 1997). An incremental convex hull algorithm can maintain the convex hull at each insertion with a trivial modification to the current convex hull without constructing the convex hull from the scratch.

**Corresponding Author's Email:* krw19870829@gmail.com
https://orcid.org/0000-0002-8933-1687

The Beneath-Beyond algorithm is one of the oldest incremental type convex hull algorithms available in literature (Kallay, 1981; Grunbaum, 1961; Preparata and Shamos, 1985). A new point insertion is carried out in three steps. In the first step, the visible facets are detected. The horizon ridges set for the point marks the visible facets boundary. If the point is above a facet then the facet is visible. As the second step, a cone of novel facets is created from the point to its horizon edges. As the third step, the visible facets are removed to construct the convex hull of the current set of points that includes the new point.

Clarkson and Shor (Clarkson and Shor, 1989) proposed a randomized incremental algorithm using dual space of half space intersections. A half space is added by computing its intersection with the polytope formed by the earlier intersections. A half space is randomly selected and added to the polytope. The list of polytope edges that intersect the half space is maintained for each unprocessed half space. All such lists together are called the conflict graph. The conflict graph is used to identify the modified edges when a half space is processed. Storing only a single modified edge for each unprocessed half space helps to reduce memory consumption to O (n). A simple search is performed through adjacent edges in order to identify the remaining modified edges. A modified version of Clarkson and Shor's algorithm was proposed later (Barber *et al.*, 1996). Instead of dual space of half spaces and polytopes, their algorithm considers the space of points and convex hulls. Therefore, the conflict graph for each facet is an outside set. Only if a point is above the facet then it is in a facet's outside set. A point not yet processed belongs to exactly one outside set as in the algorithm by Clarkson and Shor. In this variation, the furthest point of an outside set is processed instead of processing a random point.

Conflict graphs or outside sets are not maintained in other modified versions of Clarkson and Shor's algorithm. Alternatively, they store old facets of the convex hull with related links to the novel facets that replaced them. A simplex is formed from $d + 1$ of the input points and it is the beginning of the hierarchy. The visible facets chain is searched through this hierarchy in order to find a visible facet for a given point. If all facets of the existing convex hull are above the point then the point is ignored. Thus these algorithms form the convex hull. The running times of these algorithms are closer with other algorithms (Fortune, 1993).The comparison of the quick hull algorithm against the randomized incremental algorithms can be performed simply by changing the selection step of the quick hull algorithm. Furthermore, the quick hull algorithm handles less number of interior points than the randomized algorithm. Therefore, the quick hull algorithm is faster. Memory consumption of the quick hull algorithm is also lesser due to the reuse of the memory used by old facets (Barber *et al.*, 1996).

The time complexity of the quick hull algorithm for an input of size *n* with *r* processed points is O (n log r). The number of points processed by the quick hull algorithm is proportional to the number of vertices of the output. The output size may be considerably smaller than the worst case size. Therefore, output sensitivity is useful (Barber *et al.*, 1996). A two dimensional optimal output sensitive algorithm was invented with O (n log h) time complexity given that *h* is the size of the output (Kirkpatrick and Seidel, 1986). A three dimensional output sensitive algorithm was invented with optimal time complexity (Charkson and Shor, 1989). Later, randomization of this three dimensional algorithm was removed (Chazelle and Matousek, 1992).

Investigation of existing incremental convex hull algorithms suggests that the optimal time complexity to handle an insertion of a point is O (n). This paper proposes a new algorithm that takes O (h^2) time to insert a point given that there are h points on the current convex hull. The proposed algorithm is relatively simple and represents the convex hull using a set of line segments. Furthermore, the three dimensional extension of the algorithm is possible.

MATERIALS AND METHODS

An incremental type convex hull algorithm is introduced in this section. The algorithm outputs a set of line segments forming the convex hull of the point set. The edges are stored using the end points. The number of edges is stored in the variable *edges* and the edges are numbered as 0, 1…, (*edges* - 1). The variable *edges* is initialized to 0. The coordinates of the starting point and the ending point of the i^{th} edge are represented by *(sx[i], sy[i])* and *(ex[i], ey[i])* respectively. These four coordinates of the edges are stored using a set of four arrays. These four arrays together are called the edge list. When a new edge is inserted to the edge list then the variable *dead[i]* for that edge is set to 0. That means the edge is on the convex hull. However, some edges will no longer be on the convex hull due to successive growth of the convex hull. The corresponding variable *dead[i]* is set to 1 instead of deleting such edges. This means the edge no longer belongs to the convex hull. Therefore, the set of edges on the convex hull is a subset of edges from the edge list in which the variable *dead[i]* is 0.

The development of the convex hull is carried out in two phases: the generation of the primary hull and the generation of the secondary hull. Following sections describe each phase. The primary hull construction is handled using function *makePHull()* and the secondary hull construction is handled using function *makeSHull()*. A separate function *makeHull()* is used to choose appropriate construction phase. The function *makeHull()* is executed exactly once at each insertion of a data point.

The use of function *makeHull()*

The pseudo code for function *makeHull()* is given below.

```
1.  edges = 0;
2.  FUNCTION makeHull()
3.  BEGIN
4.  IF edges < 3 THEN
5.      CALL makePHull();
6.  ELSE
7.      CALL makeSHull();
8.  END IF
9.  END
```

If the number of edges in the edge list is less than 3 then the function *makePHull()* is executed (Line 5). Otherwise the function *makeSHull()* is executed (Line 7). The justification for this conditional logic is provided in the sections given below.

The use of function *makePHull()*

The pseudo code for function *makePHull()* is given below.

The primary hull is made using the first three non-collinear points. The variable *distinct* stores the number of non-collinear points and it is initially set into 0 (Line 1). The first point is included as the starting point of the 0^{th} edge (Line 5). The first point is itself the hull. Then the variable *distinct* is set to 1. The algorithm waits until a distinct point arrives (Line 8) and that point is set as the ending point of the 0^{th} edge (Line 9). Then the variable *distinct* is set to 2 and variable *edges* is set to 1.

```
1.  distinct = 0;
2.  FUNCTION makePHull()
3.  BEGIN
4.  IF distinct = 0 THEN
5.        sx[0] = x; sy[0] = y;
6.        distinct = 1;
7.  ELSE IF distinct = 1 THEN
8.          IF sx[0] != x OR sy[0] != y THEN
9.                  ex[0] = x; ey[0] = y;
10.                 distinct = 2;
11.                 edges = 1;
12.                 dead[0] = 0;
13.         END IF
14. ELSE
15.         IF sx[0] * (ey[0] - y) + ex[0] * (y - sy[0]) + x * (sy[0] - ey[0]) = 0 THEN
16.                 IF sx[0] = ex[0] THEN
17.                         IF  ey[0] > sy[0] THEN
18.                                 IF y > ey[0] THEN
19.                                         ey[0] = y;
20.                                 ELSE IF y < sy[0] THEN
21.                                         sy[0] = y;
22.                                 END IF
23.                         ELSE
24.                                 IF y > sy[0] THEN
25.                                         sy[0] = y;
26.                                 ELSE IF y < ey[0] THEN
27.                                         ey[0] = y;
28.                                 END IF
29.                         END IF
30.                 ELSE
31.                         IF ex[0] > sx[0] THEN
32.                                 IF x > ex[0] THEN
33.                                         ex[0] = x; ey[0] = y;
34.                                 ELSE IF x < sx[0] THEN
35.                                         sx[0] = x; sy[0] = y;
36.                                 END IF
37.                         ELSE
38.                                 IF x > sx[0] THEN
39.                                         sx[0] = x; sy[0] = y;
40.                                 ELSE IF x < ex[0] THEN
41.                                         ex[0] = x; ey[0] = y;
42.                                 END IF
43.                         END IF
44.                 END IF
45.         ELSE
46.                 sx[1] = x; sy[1] = y; ex[1] = sx[0]; ey[1] = sy[0];
47.                 dead[1] = 0;
48.                 sx[2] = x; sy[2] = y; ex[2] = ex[0]; ey[2] = ey[0];
49.                 dead[2] = 0;
50.                 edges = 3;
51.         END IF
52. END IF
53. END
```

The next point has two possibilities: collinear or non-collinear with the 0th edge. These two possibilities should be handled in two different ways and the following two sections describe these two techniques. The co-linearity can be tested using the area of the triangle made by *(sx[0], sy[0]), (ex[0], ey[0])* and *(x, y)* where *(x, y)* denotes the new point. If the area of the triangle is zero then these three points are collinear (Line 15). Otherwise the three points are non-collinear (Green, 1991).

1. *Next point is collinear with 0th edge (Line 16 to 44)*

The hull remains as a line segment in this situation. If the 0th edge is parallel to y-axis (Line 16) then the projection of the three points on y-axis is considered (Line 17 to 28). Otherwise, the projection on x-axis is considered (Line 31 to 43). If the novel point is on the 0th edge then no modification is needed. Otherwise, the closest vertex to the novel point should be replaced by the novel point. After this step, the next point also executes function *makePHull()* since variable *edges* < 3.

2. *Next point is non-collinear with 0th edge (Line 46 to 50)*

Two more edges are included to the edge list in this situation. Then the variable *edges* should be set to 3 (Line 50). The starting points of both 1st and 2nd edges should be new point (Line 46, Line 48). The end point of the 1st edge should be the starting point of the 0th edge. The ending point of the 2nd edge should be the ending point of the 0th edge (Line 46, Line 48).

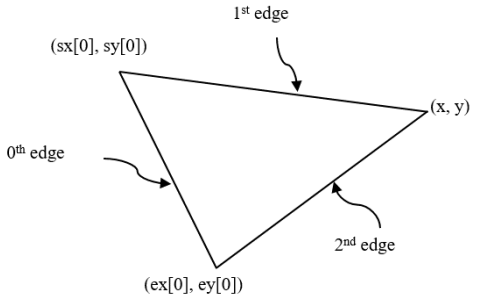


Figure 1: The primary hull.

When the number of edges in the edge list is three then the hull is called the primary hull as shown in Figure 1. According to function *makeHull()*, the new points execute function *makeSHull()* instead of function *makePHull()* after the convex hull has reached the state of primary hull.

Calculating the centroid of the primary hull

An internal point of the hull should be found after generating the primary hull. Centroid of any triangle falls inside the triangle. The centroid of the primary hull is inside the hull even through the successive growth of the hull. Therefore, the centroid of the primary hull is selected as the interior point. Calculating the centroid of the primary hull is done by using function *calCentroid()*. The relevant pseudo code is given below.

```
1.  FUNCTION calCentroid()
2.  BEGIN
3.  xc = (sx[0] + ex[0] + sx[1]) / 3;
4.  yc = (sy[0] + ey[0] + sy[1]) / 3;
5.  FOR i = 0 TO i = 2
6.      IF sx[i] = ex[i] THEN
7.              det[i] = 0;
8.              valc[i] = xc - sx[i];
9.      ELSE
10.             det[i] = 1;
11.             m[i] = (ey[i] - sy[i]) / (ex[i] - sx[i]);
12.             c[i] = sy[i] - m[i] * sx[i];
13.             valc[i] = m[i] * xc – yc + c[i];
14.     END IF
15. END FOR
16. END
```

The coordinates of the centroid of the primary hull (xc, yc) is calculated (Line 3, Line 4). Then gradient (m[i]) and y-intercept (c[i]) of each of the edges of the primary hull should be calculated (Line 5 to 15). Since there can be an edge with indeterminate gradient the *det[i]* variable is maintained. If the gradient is indeterminate then *det[i]* should be set to 0. Otherwise it should be set to 1. If an edge is parallel to y-axis then its gradient is not defined (Line 6). The aim of calculating gradient and y-intercept of edges is to calculate the value of (a * xc + b * yc + c) for the edge (If the equation of the line is [a * x + b * y + c = 0]). Here (y = m * x + c) form of the equation is used for straight lines. Then it can be reformulated as (m * x – y + c = 0). The value of (m[i] * xc + yc + c[i]) or value of (xc – sx[i]) is stored in *valc[i]* variable (Line 8, Line 13).

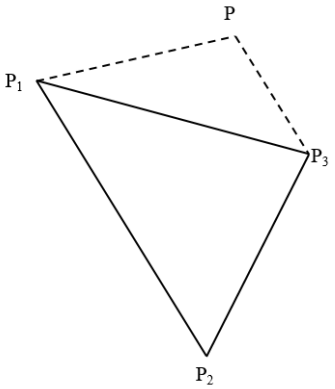


Figure 2: One edge is dead.

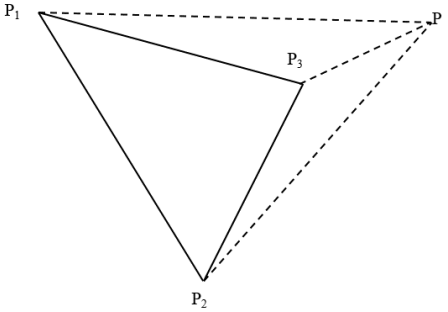


Figure 3: Two edges are dead.

The Use of Function *makeSHull()*

The pseudo code for function *makeSHull()* is given below.

```
1.  calculated = 0;
2.  FUNCTION makeSHull()
3.  BEGIN
4.  i_edges = edges;
5.  IF calculated = 0 THEN
6.      CALL calCentroid();
7.      calculated = 1;
8.  END IF
9.  FOR i = 0 TO i = i_edges – 1
10.     IF dead[i] = 0 THEN
11.         IF det[i] = 1 THEN
12.             IF valc[i] * (m[i] * x – y + c[i]) < 0 THEN
13.                 dead[i] = 1;
14.                 sx[edges] = x;      sy[edges] = y;
15.                 ex[edges] = sx[i]; ey[edges] = sy[i];
16.                 dead[edges] = 0;
17.                 edges = edges + 1;
18.                 sx[edges] = x;      sy[edges] = y;
19.                 ex[edges] = ex[i]; ey[edges] = ey[i];
20.                 dead[edges] = 0;
21.                 edges = edges + 1;
22.             END IF
23.         ELSE
24.             IF valc[i] * (x-sx[i]) < 0 THEN
25.                 dead[i] = 1;
26.                 sx[edges] = x; sy[edges] = y;
27.                 ex[edges] = sx[i]; ey[edges] = sy[i];
28.                 dead[edges] = 0;
29.                 edges = edges + 1;
30.                 sx[edges] = x; sy[edges] = y;
31.                 ex[edges] = ex[i]; ey[edges] = ey[i];
32.                 dead[edges] = 0;
33.                 edges = edges +1;
34.             END IF
35.         END IF
36.         IF dead[i] = 1 THEN
37.             FOR j = i_edges TO j = edges - 1 THEN
38.                 IF sx[j] = ex[j] THEN
39.                     det[j] = 0;
40.                     valc[j] = xc - sx[j];
41.                 ELSE
42.                     det[j] = 1;
43.                     m[j] = (ey[j] - sy[j]) / (ex[j] - sx[j]);
44.                     c[j] = sy[j] - m[j] * sx[j];
45.                     valc[j] = m[j] * xc – yc + c[j];
46.                 END IF
47.             END FOR
48.         END IF
49.     END IF
50. END FOR
51. FOR i = i_edges TO i = edges -1
52.     FOR j = i + 1 TO j = edges – 1
53.         IF ex[i] = ex[j] AND ey[i] = ey[j] THEN
54.             dead[i] = 1;
55.             dead[j] = 1;
56.         END IF
57.     END FOR
58. END FOR
59. END
```

Initial number of edges in the edge list is first assigned to variable *i_edges* (Line 4). Then method *calCentroid()* is invoked to calculate the centroid of the primary hull (Line

6). The variable *calculated* (Line 1, Line 5, Line 7) is used to avoid recalculation of the centroid. The primary hull has already been constructed using the function *makePHull()*. The next point may arrive as two possible situations as shown in Figure 2 and Figure 3 respectively. There, $P_1P_2P_3$ is the current hull and P is the novel point. If the novel point falls inside the triangle then it is ignored. If the coordinates of the novel point is (x, y) then the value of *valc[i] * (m * x – y + c)* is calculated with respect to each hull-edge (with determinate gradients) of the existing hull (Line 12). If the gradient of an edge is undefined then for those edges value of *valc[i] * (x – sx[i])* should be calculated. If that value is negative then the centroid of the primary hull and new point are on the opposite sides of the relevant hull-edge (Line 12, Line 24). The variable *dead[i]* of those edges should be set into 1.

In situation shown in Figure 2, P_1P_3 edge should set dead and in situation shown in Figure 3, both P_2P_3 and P_1P_3 edges should set dead. Once an existing hull-edge is dead, new two edges are included to the edge list as non-dead edges. Starting point of both of those two edges should be the new point. And the ending points should be the ending points of each dead edge. In Figure 2, PP_1 and PP_3 edges should be added to the edge list as non-dead edges. In Figure 3, PP_1 and PP_3 edges are included to the edge list as non-dead edges due to the death of P_1P_3, and also PP_2 and PP_3 edges are included to the edge list as non-dead edges due to the death of P_2P_3 edge. It can be observed that PP_3 edge has been added to the edge list twice (Line 13 to 21, Line 25 to 33). After inclusion of new edges, values of their *det[i], m[i], c[i],* and *valc[i]* should be calculated (Line 37 to 47).

Figure 4 is used to describe the subsequent part of the algorithm. Let $P_1P_2P_3P_4P_5P_6$ is the current hull and P is the new point. Here P_2P_3, P_3P_4, P_4P_5, P_5P_6 edges should set dead. Death of P_2P_3 leads to include PP_2 and PP_3. Death of P_3P_4 leads to include PP_3 and PP_4. Death of P_4P_5 leads to include PP_4 and PP_5. Death of P_5P_6 leads to include PP_5 and PP_6. Then the edge list contains PP_3, PP_4 and PP_5 edges each twice as still non-dead edges. These edges should set dead. This can be done by searching for the duplicate edges after each inclusion of a new point (Line 51 to 58). Thus the resultant convex hull can be generated.

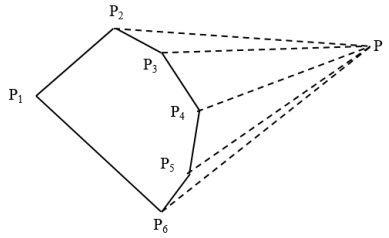


Figure 4: Updating the convex hull.

Figure 5 illustrates an example of generation of convex hull dynamically using the proposed algorithm. The dotted lines show the dead edges in the edge list. And the thick lines show the hull-edges.

Construction of the convex hull using a set of points

A sequence of random points is used to demonstrate the algorithm. Let the number of random points in the sequence be *p*. A pseudo code is given below.

```
1.  FOR i = TO p – 1
2.      x = RANDOM_NUMBER;
3.      y = RANDOM_NUMBER;
4.      CALL makeHull();
5.      DISPLAY_CONVEX_HULL;
6.      WAIT;
7.      ERASE_CONVEX_HULL
8.  END FOR
```

A random point is generated in each time (Line 2 and Line 3). The function *makeHull* is executed for each random point (Line 4). Then the constructed convex hull is displayed (Line 5) and erased (Line 7). The convex hull is displayed for a small amount of time before it is erased (Line 6). Thus the evolution of the convex hull can be seen on the display.

RESULTS AND DISCUSSION

Suppose that the convex hull of n points has already been computed and h points are on the convex hull. It takes O (h) time to test whether the novel point is inside or outside with respect to each edge of the convex hull. The new point is outside (h - 1) edges of the convex hull in the worst case. Therefore, [2 (h - 1)] edges are included to the edge list. To set dead the duplicate edges, it takes O (h^2) time. Thus, it takes O (h) + O (h^2) = O (h^2) time to insert a novel point to the existing convex hull in the worst case. The number of additional memory locations is proportional to the number of edges included to the edge list. None of the edges are deleted and instead the state of each of them is set dead. Thus, the space complexity of the proposed algorithm is O (h).

The Beneath-Beyond algorithm deletes the old facets in order to insert a new point. Therefore, the time complexity of inserting a new point is O (n) (Kallay, 1981; Grunbaum, 1961; Preparata and Shamos, 1985).

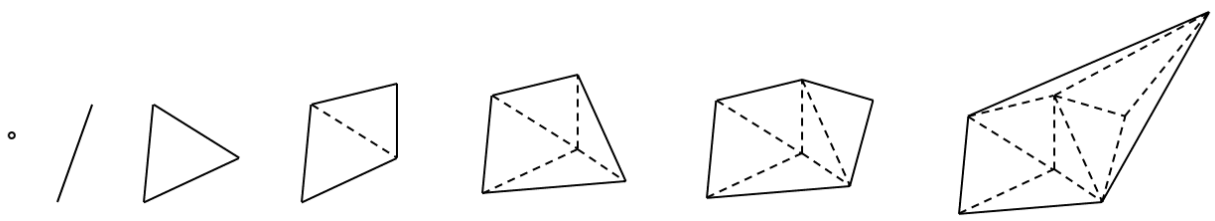


Figure 5: Illustrative example for the generation of the convex hull.

The proposed algorithm does not delete the old facets and instead the state of each facet is set dead. Thus the proposed algorithm takes only O (h^2) time to insert a new point as discussed above. Later, output sensitive algorithms were invented with O (n log h) time complexity (Barber *et al.*, 1996; Kirkpatrick and Seidel, 1986; Charkson and Shor, 1989; Chazelle and Matousek, 1992). However, these output sensitive algorithms also take O (n) time to insert a new point even though they take only O (n log h) time to construct the convex hull from the scratch. Therefore, if h < SQRT (n) then the proposed algorithm inserts a new point faster than the existing fastest algorithms.

The three dimensional extension of the proposed algorithm is possible (Wijeweera and Kodituwakku, 2018). A list of triangular facets is maintained instead of the edge list. The novel point is tested for being inside or outside each facet of the existing convex hull. The facets on the existing convex hull to which the new point is outside are set dead. Death of a single facet on the current convex hull leads to insertion of three new facets to the facet list. The duplicate facets are removed. Thus, the new convex hull can be constructed.

CONCLUSION

An incremental type planar convex hull algorithm was proposed in this paper. The algorithm takes O (h^2) time and O (h) space to insert a novel point. The proposed approach is faster than the available fastest algorithms when h < SQRT (n). Furthermore, the three dimensional extension of the proposed algorithm is possible.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers as their feedback helped to extensively improve the quality of this research article.

DECLARATION OF CONFLICT OF INTEREST

There are no conflicting interests.

REFERENCES

Akl, S.G. and Toussaint, G.T. (1978). A fast convex hull algorithm. *Information Processing Letters* **7**(5): 219-222. DOI: https://doi.org/10.1016/0020-0190(78)90003-0.

Akl, S.G. and Toussaint, G.T. (1978). Efficient convex hull algorithms for pattern recognition applications. *Proceedings of 4th International Joint Conference on Pattern Recognition, Kyoto, Japan*, Pp. 483-487.

Avis, D. (1979). *On the Complexity of Finding the Convex Hull of a Set of Points*. Technical Report, School of Computer Science, McGill University.

Bayer, V. (1999). *Survey of algorithms for the convex hull problem*, Department of Computer Science, Oregon State University, United States, 20. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.8677 (May 24, 2018).

Bently, J.L. and Shamos M.I. (1978). Divide and conquer for linear expected time. *Information Processing Letters* **7**(2): 87-91. DOI: https://doi.org/10.1016/0020-0190(78)90051-0.

Bhattacharya, B.K. and Toussaint, G.T. (1981). *A Time and Storage Efficient Implementation of an Optimal Planar Convex Hull Algorithm*. Technical Report, School of Computer Science, McGill University.

Boas, P.V.E. (1980). On the O (n log n) lower bound for convex hull and maximal vector determination. *Information Processing Letters* **10**(3): 132-136. DOI: https://doi.org/10.1016/0020-0190(80)90064-2.

Devroye, L. and Toussaint G.T. (1981). A note on linear expected time algorithms for finding convex hulls. *Computing* **26**(4): 361-366, DOI: https://doi.org/10.1007/BF02237955.

Eddy, W.F. (1977). A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software* **3**(4): 398-412. DOI: https://doi.org/10.1145/355759.355768.

Graham, R.L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters* **1**(4): 132-133. DOI: https://doi.org/10.1016/0020-0190(72)90045-2.

Green, S.L. (1991). *Advanced Level Pure Mathematics*. North Point, Hong Kong: University Tutorial Press, 37-46.

Jarvis, R.A. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters* **2**(1): 18-21, DOI: https://doi.org/10.1016/0020-0190(73)90020-3.

Kallay, M. (1984). The complexity of incremental convex hull algorithms in R^d. *Information Processing Letters* **19**(4): 197, DOI: https://doi.org/10.1016/0020-0190(84)90084-X.

Kirkpatrick, D.G. and Seidel, R. (1986). The ultimate planar convex hull algorithm. *SIAM Journal on Computing* **15**(1): 287-299. DOI: https://doi.org/10.1137/0215021.

O'Rourke, J. (1997). *Computational Geometry in C*. England, United Kingdom: Cambridge University Press, 63-100.

Preparata, F.P. (1979). An optimal real-time algorithm for planar convex hulls. *Communications of the ACM* **22**(7): 402-405. DOI: https://doi.org/10.1145/359131.359132.

Preparata, F.P. and Hong, S.J. (1977). Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM* **20**(2): 87-93. DOI: https://doi.org/10.1145/359423.359430.

Shamos, M.I. (1978). *Computational Geometry*. Ph.D. Thesis, Yale University, United States.

Shamos, M.I. (1981). Personal communication to G. Toussaint as described in Toussaint, G.T. computational geometric problems in pattern recognition. In: J. Kittler, K.S. Fu, L.S. Pau (Eds.), *Pattern Recognition Theory and Applications*, NATO Advanced Study Institute, Oxford University Pp. 73-91.

Toussaint, G.T., Akl S.G. and Devroye, L.P. (1978). *Efficient Convex Hull Algorithms for Points in Two and More Dimensions*. Technical Report, School of Computer Science, McGill University.

Wijeweera, K.R. and Kodituwakku, S.R. (2018). A simple and efficient incremental convex hull algorithm in 3D space. *Proceedings of the Fifth Ruhuna International Science and Technology Conference*, Pp. 47.

Yao, A.C. (1981). A lower bound to finding convex hulls. *Journal of the ACM* **28**(4): 780-789. DOI: https://doi.org/10.1145/322276.322289.

Grunbaum, B. (1961). Measure of symmetry for convex sets. *Proceedings of the Seventh Symposium in Pure Mathematics of the American Mathematical Society, Symposium on Convexity*, Pp. 233-270.

Kallay, M. (1981). *Convex Hull Algorithms in Higher Dimensions*. Unpublished manuscript, Department of Mathematics, University of Oklahoma, Norman, Okla.

Preparata, D. F. and Shamos M. (1985). *Computational Geometry. An Introduction*, Berlin, Germany: Springer-Verlag, 95-149.

Clarkson, K. and Shor, P. (1989). Applications of random sampling in computational geometry. *Discrete & Computational Geometry* **4**(5): 387-421. DOI: https://doi.org/10.1007/bf02187740.

Barber, C.B., Dobkin, D.P. and Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Transactions of Mathematical Software* **22**(4): 469-483. DOI: https://doi.org/10.1145/235815.235821.

Fortune, S. (1993). Computational geometry. In: Directions in Geometric Computing, R. Martin (Eds.), *Information Geometers*, Winchester, U. K, pp. 97-112.

Chazelle, B. and Matousek, J. (1992). *Randomizing an output-sensitive convex hull algorithm in three dimensions*. Technical Report, Princeton University, Princeton, N. J.