A Simple Plane Sweep Convex Layers Algorithm

Raimi Rufai* Dana Richards†

Abstract

Existing optimal sequential algorithms for the convex layers problem tend to be quite involved. In this paper, we present a simple plane-sweep algorithm for the convex layers problem. The algorithm runs in $O(p_e kn \log n)$ time and $O(n)$ space where k is the number of layers and $p_e \in (0, 1)$ is an input dependent factor. For a point set that uniformly distributed in a square, $p_e = \Theta(1/k)$.

1 Introduction

One way of computing the convex layers of a point set P is by taking its convex hull to obtain its first layer L_1. These points are then discarded and the convex hull of the remaining points are taken to obtain the second layer L_2. Those are then discarded and we continue this process until there are no more points left. So, in general a point p belongs to layer L_i, if it lies on the convex hull of the point set $P - \bigcup_{j=1}^{i-1}\{L_j\}$.

The *convex layers*, $\mathbb{L}(P) = \{L_1, L_2, \cdots, L_k\}$, of a set P of $n \geq 3$ points is a partition of P into $k \leq \lceil n/3 \rceil$ disjoint subsets L_i, $i = 1, 2, \cdots, k$ called *layers*, such that each layer L_i is an ordered[1] set of the hull vertices of the set $\bigcup_{j=i,\cdots,k}\{L_j\}$. Figure 1 below shows the convex layers for a set of 15 points with 3 layers.

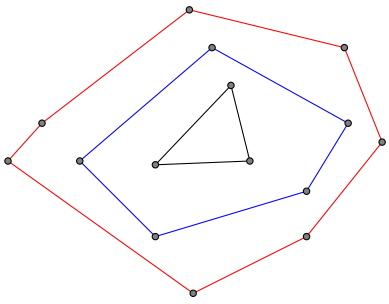

Figure 1: Convex Layers

*Department of Computer Science, George Mason University, rrufai@gmu.edu

†Department of Computer Science, George Mason University, richards@gmu.edu

[1]One convention is to have the points sorted in the counterclockwise order starting with the one with the smallest x coordinate, breaking ties by choosing the one with the smallest y.

Thus, the outermost layer L_1 coincides exactly with the convex hull of P, $\text{conv}(P)$. Next, we define the *convex layers problem*. The *convex layers problem* is to compute $\mathbb{L}(P)$ for a given point set P.

In this paper, we present a simple plane sweep algorithm for the convex layers problem. Section 2 relates relevant literature. Section 3 presents our proposed algorithm. Section 4 concludes the paper and speculates on future work.

2 Related Work

A brute-force solution to the convex layers problem is obvious – construct each layer L_i as the convex hull of the set $P - \bigcup_{j<i} L_j$ using some suitable convex hull algorithm. The brute-force algorithm will take $O(kn \log n)$ time where k is the number of layers. It essentially computes one convex layer at a time by peeling off the points on that layer. Intuitively, one can visualize these algorithms as peeling off the layers one at a time from some geometric structure such as a point set, a Delaunay triangulation, or a Voronoi diagram. This *peeling* approach is reminiscent of many convex layers algorithms. Another general approach to this problem is the *plane-sweep* paradigm. We shall review algorithms in both categories below.

2.1 Peeling-based Techniques

One of the earliest works that takes this approach is Green and Silverman [7]. Their algorithm is a repeated invocation of QuickHull to extract the convex layers, one layer per invocation. This algorithm runs in $O(n^2)$ worst-case time.

Overmars and van Leeuwen [12] proposed an algorithm for this problem that runs in $O(n \log^2 n)$ based on a fully dynamic data structure for maintaining a convex hull under arbitrary deletions and insertion of points. Each of these update operation takes $O(\log^2 n)$ time, since constructing the convex layers can be reduced to inserting all the points into the data structure in time $O(n \log^2 n)$, marking points on the current convex hull and deleting them off and then repeating this for the next layer. Since each point is marked exactly once and

deleted exactly once in the life of the algorithm, these steps together take no more than $O(n \log^2 n)$ time.

Chazelle [4], using a deletion-only semi-dynamic hull graph data structure similar to that Overmars and van Leeuwen [12] gave an optimal algorithm for this problem that runs in $O(n \log n)$ time and $O(n)$ space, both of which are optimal. The deletion-only semi-dynamic data structure of Hershberger and Suri [10] can also be used to achieve the same bounds. Using the fully dynamic parametric heap data structure of Brodal and Jacob [2], which admits an amortized $O(\log n)$ per update operation, an $O(n \log n)$ peeling algorithm for the convex layers can been devised.

All of these latter results are not easy to implement as they use complicated data structures.

2.2 Plane-sweep Technique

The first algorithm on record that uses this technique is a modification of Jarvis march proposed by Shamos [14]. The algorithm works by doing a radial sweep, changing the pivot along the way, but does not stop after finding all the points on the convex hull (L_1). It proceeds with another round of Jarvis march that excludes points found to belong to previously computed layers, and then another round, and so on until all the layers have been computed. The algorithm runs in $O(n^2)$.

Naturally, one might start to wonder if Chan's modification of Jarvis march [3] that uses a grouping trick can lead to an optimal solution to this problem. This is exactly the approach taken by Nielsen [11] to obtain yet another optimal algorithm for the convex layers problem. Nielsen's algorithm is output-sensitive in that it can be parameterized to compute a given number of layers. It runs in $O(n \log H_k)$ time where H_k is the number of points appearing on the first k layers.

2.3 Other results

Dalal [5] showed that the expected number of convex layers for a set of n points uniformly and identically distributed within a smooth region such as a d-dimensional ball is instead $\Theta(n^{2/(d+1)})$.

The envelope layers problem [9] and the multi-list layering problem [6] have been shown to be P-complete. It is still not known whether the convex layers problem belongs to the class NC [1, 9] – the class of problems that have parallel algorithms that run in polylogrithmic time given a polynomial number of processors. Dessmark et al. [6] reported a reduction of the convex layers problem to the multi-list layering problem, but it is the inverse reduction that is needed to resolve the status of the convex layers problem.

3 Results

This section describes Algorithm ConvexLayers, a plane sweep convex layers construction algorithm and the data structures used in the algorithm. We first describe the data structures in §3.1 and then the workings of the algorithm in §3.2 and §3.3.

Input: A point set P, and parameter k for the maximum number of layers to return.
Output: The convex layers of P, $\mathbb{L}(P)$.

```
ConvexLayers(P = {p_i : i = l ··· n}, k)
1  if n < 4
2     then L ← sort(P)
3     else T ← ComputeUpperLayers(P, k)
4          B ← ComputeLowerLayers(P, k)
5          L ← mergeLayers(T, B)
6  return L
```

Figure 2: Algorithm ConvexLayers

Algorithm ConvexLayers consists of these high-level steps:

Build upper layers. Sweep the input point set P from top to bottom to build downward-facing partial layers.

Build lower layers. Sweep P from bottom to top to construct upward-facing partial layers.

Merge. Merge these two partial layers and throw away spurious layers to obtain $\mathbb{L}(P)$.

3.1 Data Structures

The data structures used by the algorithm include a status structure T that dynamically maintains the upper layers as points are encountered.

Status structure. The top-to-bottom sweep status structure T is an array of linked lists that is updated as the sweep line travels down the point set. Thus, at the end of the sweep, T_i contains a linked list of points on the upper layer i, ordered counterclockwise (i.e. from right to left in this case).

There is an analogous bottom-to-top sweep status structure B as well. Note that the merge step might nip off some of the layers at one or both ends and eliminate some layers entirely.

Event Queue. A priority queue Q of events sorted according the y-coordinates of the associated points.

Events. There are two kinds of events that are generated by the algorithm:

New Point Event. A new point is encountered by the sweep line.

Eviction Event. A chain of points is evicted from a layer.

3.2 Building Upper Layers

Algorithm ComputeUpperLayers sweeps the point set from top to bottom along the y-axis toward the line $y = -\infty$. As we sweep through, we update the status structure T. When the sweep line crosses a new point p, we do a search to find the layer L_i that p belongs to and insert it.

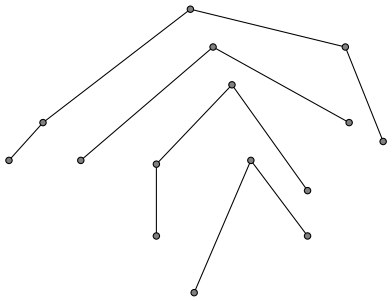

Figure 3: Upper Layers

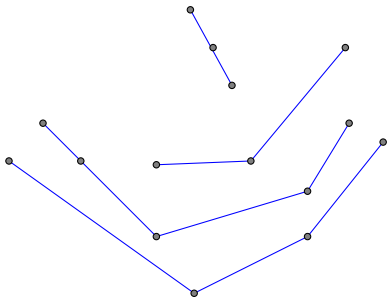

Figure 4: Lower Layers

After the sweep line has traversed all the points, we will end up with a set of upper hull layers, which may include some unnecessary layers that will later be eliminated by the merge step. We similarly construct the lower layers. We then merge them together to obtain the convex layers of P.

When a new point p is encountered, we do a search of all the intervals spanned by currently known layers to decide what layer L_i p should go into. This search can be done as follows.

Input: A point set P, and parameter k for the maximum number of layers to return.
Output: The convex layers of P, $\mathbb{L}(P)$.
ComputeUpperLayers(P, k)

```
1  Q ← ∅
2  for each p in P
3  do insert NewPointEvent(p) into Q
4  T = ()
5  while Q not empty
6  do
7     e = Q.head
8     if e is a NewPointEvent
9       then l ← LayerSearch(e.point, T.intervals)
10           if l ≤ k
11             then INSERT(e, T, l)
12      else  remove e.chain from T_{e.layer−1}
13           if e.layer ≤ k
14             then INSERT(e, T, e.layer)
15    return T
```

Figure 5: Algorithm ComputeUpperLayers

Search for layer to insert p into . Each layer L_i is associated with two intervals $X_i = (\check{x}, \hat{x})$ and $Y_i = (\check{y}, \hat{y})$ correponding to the orthogonal projections of L_i on the x- and y-axes respectively. The following algorithm computes in time $O(\log |\mathbb{L}|)$ the layer a point p encountered during a sweep should go into.

Input: A point p to be inserted, H the current number of layers, and the set of layer intervals to search.
Output: The layer i to which p belongs.
LayerSearch$(p, H, \{X_i = (p_i, q_i) : i = l \cdots h\})$

```
1  if l = h = 1
2     then return 1
3  if l = h = H
4     then return H + 1 ▷ Create a new layer
5  m ← ⌊(l + h)/2⌋
6  if (x(p) ∉ X_m(p_m, q_m))
7     and  (x(p) ∈ X_{m−1}(p_{m−1}, q_{m−1})
8     then return m
9  if x(p) ∈ X_m(p_m, q_m)
10    then return LayerSearch(p, H,
11           {X_i = (p_i, q_i) : i = m + 1 ··· h})
12    else  return LayerSearch(p, H,
13           {X_i = (p_i, q_i) : i = l ··· m − 1})
```

Figure 6: Algorithm LayerSearch

Let L be a polygonal chain. A point $q \in L$ is an anchor

point for another point $p \notin L$ if q is the end point of the tangent segment connecting p to L.

Insertion of p into the layer L_i involves computing the new edge pq introduced into L_i by p, where q is the anchor point of p in L_i. The anchor point q can be found by doing a binary search on L_i. Denote the chain of vertices in L_i below this segment by C_i. This can result in one of two cases:

Case 1 (C_i is empty) *In this case, we simply insert p into L_i and we are done. Given the anchor point q, this can be done in constant time.*

Case 2 (C_i is not empty) *If this chain of vertices C_i is not empty, an eviction event e is generated for C_i. The event e encapsulates C_i – the chain of points evicted – and the target layer they should go, which in this case is simply layer $i+1$. Event e also has reference h_i to the head of the chain, the point adjacent to the anchor point q in L_i from which the chain was evicted. The head of a chain is always the highest point in the chain* [2]*. This chain is immediately inserted into layer L_{i+1} by essentially inserting its head. Potentially, this might also result in an eviction triggered in L_{i+1}. In the worst case, a cascade of $O(k-i)$ evictions can be triggered, one for every existing layer $j > i$. For each target layer L_j, this step takes $O(\log |L_j|)$ to find an anchor point for h_j. Thus, the entire step takes $O(\sum_{j=i+1}^{k} \log |L_j|) = O((k-1)\log n)$, given the initial anchor point q.*

The key operations in the algorithm are the following:

Layer search. This takes $O(\log |\mathbb{L}|)$, as discussed above.

Insertion into a layer i. This takes $O(\log(|L_i|))$ time because of the binary search to locate the anchor point for Case 1. In Case 2, this takes $O(\sum_{j=i}^{k} \log |L_j|) = O(k \log n)$. Thus, this step takes $O(\log(|L_i|) + p_e \sum_{j=i+1}^{k} \log |L_j|) = O(p_e k \log n)$ expected time, where p_e is the probability of an eviction.

One crucial question that still remains is how to compute the probability p_e above.

1. When a point p causes a chain C_i to be evicted, combinatorial changes to the hull layers are limited to layers i and below. Shallower layers are unaffected.

2. Let $q = (q_x, q_y)$ be the anchor point for $p = (p_x, p_y)$. Let r be the point (q_x, p_y). An eviction occurs with the addition of p if and only if the triangle defined by the three points (p, q, r) contains a vertex of L_i. An example is the shaded triangle in Figure 7.

If P is uniformly distributed in a unit square, the area of this triangle will be roughly $p_e = \Theta(\frac{1}{k})$. Thus, for such a distribution, the algorithm is optimal. It will be interesting to explore the behavior of p_e with different distributions of P.

3.2.1 Eviction Probability p_e

Let $P = \{p_0, p_2, \cdots, p_{n-1}\}$ be a point set in general position that is independently and uniformly distributed in the unit square. Assume these points are sorted by decreasing y coordinates, so that $p_0 = \text{argmax}_y P$ and $p_{n-1} = \text{argmin}_y P$. Consider an arbitrary point $p \in P$ encountered during the course of the sweep algorithm.

Lemma 1 *Let $P \subset \mathbb{R}^2$ be a set of n points drawn independently and uniformly from any a disk. The probability that a point p belongs to a specific layer i is given by $\mathbb{P}(p \in L_i) = O(n^{-2/3})$.*

Proof. Previous work [5, 8, 15] has shown that the number of vertices of $\text{conv}(P)$ is $\Theta(n^{1/3})$ and the number of layers of P is $\Theta(n^{2/3})$.

This implies that each layer L_i has $O(n^{1/3})$ vertices, for every $i = 1, 2, \ldots, k$. Thus, the probability that an arbitrary point $p \in P$ belongs to a specific layer i is given by:

$$\mathbb{P}(p \in L_i) = O(n^{1/3})/n = O(n^{-2/3}). \tag{1}$$

□

Dalal [5] showed that the above result applies more generally to any fixed, bounded, full-dimensional shape with a nonempty interior, since such a shape can contain a disk and thus cannot contain fewer layers than a disk. Thus, one would expect the Lemma 1 to apply also to a square, except for the first few layers.

Denote by L_i^p the i-th layer just before encountering point p. Given that an arbitrary point p belongs in L_i and that its anchor point in L_i is the point q, the probability that p will result in an eviction is simply the probability that the right triangle $\Delta(pq)$ below the edge pq contains some point of L_i^p. This is the same as:

$$\begin{aligned} p_e &= \mathbb{P}(p \in L_i) \cdot \mathbb{P}(q \in L_i) \cdot \mathbb{E}[\text{area}(\Delta(pq))] && (2)\\ &= O(n^{-4/3}) \cdot \mathbb{E}[\text{area}(\Delta(pq))] && (3)\\ &= O(n^{-4/3}) && (4) \end{aligned}$$

where $\Delta(pq)$ denotes the triangle below the edge pq (see Figure 7 below).

[2] This is true when computing upper hulls. When computing lower hulls, the head of the chain will always be the lowest point in the chain

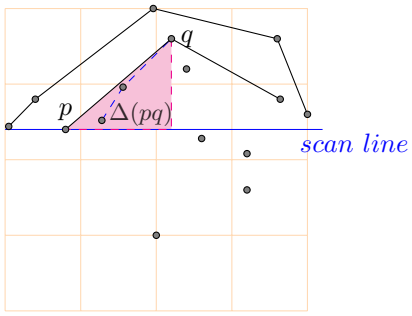

Figure 7: Eviction Triangle

Line 4 follows from previous work by several authors [16, 13], the above simplifies to:

$$\mathbb{E}[\text{area}(\Delta(pq))] = \frac{11}{144} = O(1) \tag{5}$$

3.2.2 Points from a k-gon

Alternatively, in the case of a polygon with k sides, one can argue as follows.

Lemma 2 *The probability that a point p belongs to the layer i is given by $\mathbb{P}(p \in L_i) = O(\log^{i+1} n/n)$.*

Proof. Let $P \subset \mathbb{R}^2$ be a set of n points drawn independently and uniformly from a k-gon. Previous work [8] has shown that the number of vertices of $\text{conv}(P)$ is $\Theta(k \log n)$. Let P_{i+1} denote the points that remain after - off the first i layers, so that $P_0 = P$ and $P_i = P \setminus \bigcup_{j<i} L_j$. Note that P_0 is wholly contained in a polygon with k sides, and the expected complexity of $\text{conv}(P_0)$ is $O(k \log n)$. Since P_1 is also wholly contained in $\text{conv}(P_0)$, similarly the expected complexity of $\text{conv}(P_1)$ is $O(|\text{conv}(P_0)| \log |P_1|) = O(k \log n \log |P_1|)$. Note that the points P_{i+1} are also independently and uniformly distributed within $\text{conv}(P_i)$. Thus, we obtain the following expression for the complexity of $\text{conv}(P_i)$:

$$\begin{aligned}
|L_i| &= |\text{conv}(P_i)| && (6)\\
&= O(k \cdot \Pi_{j=0}^{i} \log |P_j|) && (7)\\
&= O(k \cdot \Pi_{j=0}^{i} \log |P_0|) && (8)\\
&= O(k \cdot \log^{i+1} |P_0|) && (9)\\
&= O(k \cdot \log^{i+1} n) && (10)
\end{aligned}$$

Thus, it follows that $\mathbb{P}(p \in L_i) = O(\log^{i+1} n/n)$. □

Using the Lemma 2, we obtain the eviction probability as follows.

$$\begin{aligned}
p_e &= \mathbb{P}(p \in L_i) \cdot \mathbb{P}(q \in L_i) \cdot \mathbb{E}[\text{area}(\Delta(pq))] && (11)\\
&= O(\log^{i+1} n/n) \cdot O(\log^{i+1} n/n) \cdot \mathbb{E}[\text{area}(\Delta(pq))] && (12)\\
&= O(\log^{2i+2} n/n^2) && (13)
\end{aligned}$$

Theorem 3 *The eviction probability $p_e = O(1/k)$*

3.3 Merge Layers

Algorithm MERGELAYERS iterates through the four half-layers and merges them into one full layer as follows. The i-th layer is obtained from i-th upper, and lower layers as follows. The left intersection point and the right intersection point are used to trim the upper and lower layers to size.

A layer L_i is empty if either of the following condition holds:

- the lowest point on the i-th lower layer has a higher y coordinate than the highest point on the i-th upper layer.

When an empty layer is found, the algorithm stops and then returns the layers merged so far.

Input: A list of upper layers $\mathbb{T}$, and lower layers $\mathbb{B}$.
Output: The merged convex layers, $\mathbb{L}$.

```
MERGELAYERS(𝕋, 𝔹)
1  𝕃 ← ()
2  k ← NONTRIVIALLAYER(𝕋, 𝔹)
3  for i ← 1 to k
4  do L ← COMPUTEMERGEDLAYER(T_i, B_i)
5     if |L| > 0
6        then 𝕃 ← APPEND(𝕃, L)
7
8  return 𝕃
```

Figure 8: Algorithm MERGELAYERS

Input: An upper layer T,and a lower layer B.
Output: The merged layer L.

```
COMPUTEMERGEDLAYER(T, B)
1  L ← ()
2  li ← INTERSECTION(T, B)
3  ri ← INTERSECTION(B, T)
4  L ← APPEND(L, CLIP(T, li, ri))
5  L ← APPEND(L, CLIP(B, ri, li))
6  return L
```

Figure 9: Algorithm COMPUTEMERGEDLAYER

The function CLIP(c, p_1, p_2) trims its input chain of vertices c by discarding all the points preceding p_1 and all points succeeding p_2.

Input: A list of upper layers $\mathbb{T}$, and lower layers $\mathbb{B}$.
Output: The number of non-trivial layers, k.

```
nonTrivialLayer(𝕋, 𝔹)
1  k ← min(|𝕋|, |𝔹|)
2  for i ← k downto 1
3  do if y(highest(T_i)) ≥ y(lowest(B_i))
4        then return i
5
6  return k
```

Figure 10: Algorithm nonTrivialLayer

Algorithm mergeLayers clearly takes $O(n)$ time. Thus, the entire algorithm take $\Theta(n \log n)$ time.

4 Conclusion

While the algorithm presented is in general not worst-case optimal, it clearly is a simpler algorithm than known worst-case optimal ones. It would be interesting to further explore the behavior of the algorithm under different input distribution. It still remains open whether a simple-to-code, yet optimal algorithm for the convex layers problem exists.

References

[1] M. Atallah, P. Callahan, and M. Goodrich. P-complete geometric problems. In *SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 317–326, New York, NY, USA, 1990. ACM.

[2] G. Brodal and R. Jacob. Dynamic planar convex hull. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 617–626. IEEE, 2002.

[3] T. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, April 1996.

[4] B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Information Theory*, 31:509–517, 1985.

[5] K. Dalal. Counting the onion. *Random Struct. Algorithms*, 24(2):155–165, 2004.

[6] A. Dessmark, A. Lingas, and A. Maheshwari. Multilist layering: complexity and applications. *Theor. Comput. Sci.*, 141(1-2):337–350, 1995.

[7] P. Green and B. Silverman. Constructing the convex hull of a set of points in the plane. *Computer Journal*, 22:262–266, 1979.

[8] S. Har-Peled. On the expected complexity of random convex hulls. *CoRR*, abs/1111.5340, 2011.

[9] J. Hershberger. Upper envelope onion peeling. *Comput. Geom. Theory Appl.*, 2(2):93–110, 1992.

[10] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics*, 32(2):249–267, 1992.

[11] F. Nielsen. Output-sensitive peeling of convex and maximal layers. *Inf. Process. Lett.*, 59(5):255–259, 1996.

[12] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166 – 204, 1981.

[13] J. Philip. The area of a random triangle in a square. Technical report, Royal Inst. of Tech., Stockholm (Sweden), Dept. of Mathematics, Jan 2010.

[14] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, Oct. 1990.

[15] A. Rényi and R. Sulanke. über die konvexe hülle von n zufällig gewählten punkten. *Probability Theory and Related Fields*, 2:75–84, 1963. 10.1007/BF00535300.

[16] M. Trott. The area of a random triangle. *The Mathematica Journal*, 7:189–198, 1999.