Computing the convex hull of a planar point set

Michiel Smid*

September 4, 2003

1 Introduction

We consider one of the oldest and most extensively studied problems in computational geometry: Constructing the convex hull of a finite set of points in the plane.

Definition 1 A subset T of $\mathbb{R}^2$ is called *convex*, if for any two points p and q in T, the line segment joining p and q is entirely contained in T.

Definition 2 Let S be a finite set of points in the plane. The *convex hull* of S is the smallest (with respect to the relation $\subseteq$) convex set that contains S.

What do we mean by constructing the convex hull of S? It is clear that the boundary of the convex hull is a convex polygon whose *vertices* are points of S, and whose *edges* are line segments joining pairs of points of S. We denote this convex polygon by $CH(S)$.

Convex hull problem: Given a set S of n points in the plane, compute the vertices of $CH(S)$, sorted in counterclockwise order.

In Figure 1, the set S consists of thirteen points. The output of a convex hull algorithm should be the list $(p_1, p_2, p_3, p_4, p_5, p_6)$. We remark that the list storing the vertices of $CH(S)$ can start with an arbitrary vertex. In the example, the list $(p_3, p_4, p_5, p_6, p_1, p_2)$ would also be a valid output.

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: `michiel@scs.carleton.ca`.

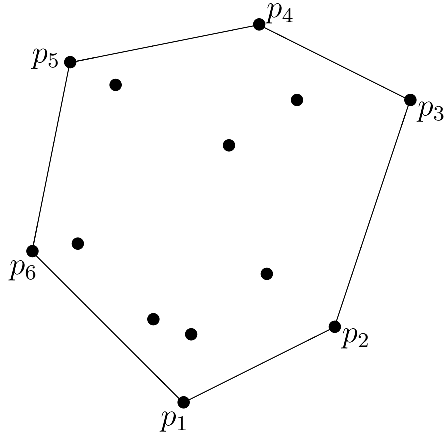


Figure 1: *An example: thirteen points, six of which determine the convex hull.*

Remark 1 If pq is an edge of the convex hull of S, and if this edge contains a point r of $S \setminus \{p, q\}$, then, by definition, r is not a convex hull vertex.

Exercise 1 Prove that constructing the convex hull is at least as hard as sorting. To be more precise, prove the following: Let $T(n)$ be the worst-case running time of an arbitrary convex hull algorithm. Then we can sort n real numbers in $T(n) + O(n)$ time.

Since in most standard models of computation, sorting takes $\Omega(n \log n)$ time in the worst case, the convex hull problem has the same lower bound in these models.

Exercise 2 Let S be a finite set of points in the plane. For each point $p \in S$, let $F(p)$ be that point of S that is furthest away from p. Let V be the set of vertices of the convex hull of S.
(i) Prove that $\{F(p) : p \in S\} \subseteq V$.
(ii) Is $V \subseteq \{F(p) : p \in S\}$?

In these notes, we will see several algorithms that solve the convex hull problem.

2 Jarvis' March

This algorithm, invented by Jarvis in 1973, is very intuitive. It is also known as the *gift wrapping method*, because the algorithm proceeds from one convex hull vertex to the next one, as if we were wrapping a sheet of paper around the set of points. The algorithm works as follows.

We start by finding a point p_1 of S that is guaranteed to be a vertex of the convex hull. We take for p_1 the lowest point of S. If S contains several points with minimum y-coordinate, then we take the leftmost of these points. Clearly, p_1 can be found in $O(n)$ time.

Since p_1 is a convex hull vertex, there must be another point p_2 in S, such that p_2 is a convex hull vertex, and p_1p_2 is a convex hull edge. Of course, there are two such points[1]. We take for p_2 that point such that p_1p_2 is a "counterclockwise" edge.

How do we find p_2? Let ℓ_1 be the horizontal line through p_1. Then p_2 is the first point that is hit when we rotate ℓ_1 in counterclockwise order around p_1. That is, for each point $q \in S \setminus \{p_1\}$, let α_q be the angle between ℓ_1 and the segment p_1q. Hence, α_q is the angle by which we have to rotate ℓ_1 in counterclockwise order around p_1 until it hits p_1q. Note that $0 \leq \alpha_q < 2\pi$. Then p_2 is that point of $S \setminus \{p_1\}$, for which this angle is minimum. If there is more than one point whose α-angle is minimum, then p_2 is that point among them having maximum distance from p_1.

Hence, given p_1, we can find the next convex hull vertex p_2, by considering all points $q \in S \setminus \{p_1\}$, and selecting that point for which α_q is minimum. That is, we can find p_2 in $O(n)$ time.

Of course, we now proceed in the same way: Let ℓ_2 be the line through the points p_1 and p_2. For each point $q \in S \setminus \{p_2\}$, let α_q be the angle between ℓ_2 and the segment p_2q. (Now α_q is the angle by which we have to rotate ℓ_2 in counterclockwise order around p_2 until it hits p_2q.) Then p_3—the next convex hull vertex—is that point whose α-angle is minimum. Hence, p_3 can be found in $O(n)$ time.

We continue computing convex hull vertices $p_4, p_5, \ldots$, until we are back in p_1. More precisely, if $p_{h+1} = p_1$, then we are done and output the list

$$(p_1, p_2, \ldots, p_h).$$

A complete description of Jarvis' March is given in Figure 2. See also Figure 3 for an illustration.

Algorithm *Jarvis_March*(S)
(* S is a set of n points in $\mathbb{R}^2$ *)
p_1 := lowest point of S;
ℓ_1 := horizontal line through p_1;
(* rotate ℓ_1 in counterclockwise order around p_1 until it hits
  at a point of S *)
for each $q \in S \setminus \{p_1\}$
do α_q := angle between ℓ_1 and the segment $p_1 q$
endfor;
q := point of $S \setminus \{p_1\}$ for which α_q is minimum;
$p_2 := q$;
ℓ_2 := line through p_1 and p_2;
$k := 2$;
while $p_k \neq p_1$
do (* rotate ℓ_k in counterclockwise order around p_k until it
    hits at a point of S *)
  for each $q \in S \setminus \{p_k\}$
  do α_q := angle between ℓ_k and the segment $p_k q$
  endfor;
  q := point of $S \setminus \{p_k\}$ for which α_q is minimum;
  $p_{k+1} := q$;
  ℓ_{k+1} := line through p_k and p_{k+1};
  $k := k + 1$
endwhile;
output the points $p_1, p_2, \ldots, p_{k-1}$

Figure 2: *Jarvis' March computes the convex hull of a set of points.*

It should be clear that this algorithm is correct. What is the running time? The first convex hull vertex p_1 is found in $O(n)$ time. Given $p_1, p_2, \ldots, p_k$, we find the next convex hull vertex p_{k+1} in $O(n)$ time. Hence, if we denote the number of convex hull vertices by h, then the running time

[1] Are there always two such points?

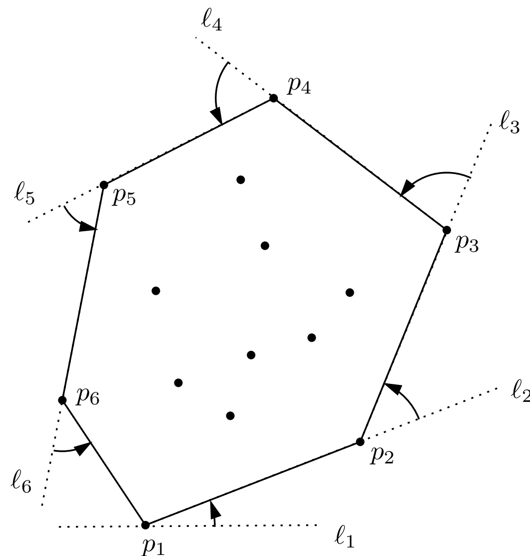


Figure 3: *Illustrating Jarvis' March.*

---

of the entire algorithm is

$$O\left(\sum_{k=1}^{h+1} n\right) = O(n(h+1)) = O(nh).$$

Theorem 1 *Let S be a set of n points in the plane and let h be the number of vertices of its convex hull. Jarvis' March computes the convex hull of S in $O(nh)$ time.*

Since h can take any value between 2 and n, the worst-case running time of Jarvis' March is $\Theta(n^2)$.

Exercise 3 If h is a constant, then Jarvis' March runs in $O(n)$ time. Is this a contradiction to the lower bound mentioned after Exercise 1?

We still have to fill in some details. In our algorithm, angles are minimized. How do we do that?

Let a and b be two consecutive (counterclockwise) convex hull vertices, and let ℓ be the line through them. We make ℓ a directed line, by giving it

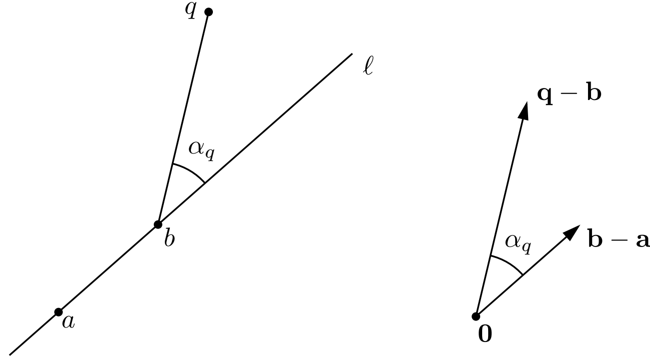


Figure 4: *How to compute the angle α_q.*

---

the direction from a to b. Note that all points of S are on or to the left of this directed line ℓ.

For any point q of $S \setminus \{b\}$, let α_q be the angle between ℓ and the segment bq. Our algorithm computes that point q for which α_q is minimum. We can do this, by computing all angles α_q explicitly and then selecting the smallest one.

Let us see how we can compute the angle α_q. (Refer to Figure 4.) First note that α_q is the angle between the vectors $\mathbf{b} - \mathbf{a}$ and $\mathbf{q} - \mathbf{b}$. We know from linear algebra that

$$(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{q} - \mathbf{b}) = \|\mathbf{b} - \mathbf{a}\| \times \|\mathbf{q} - \mathbf{b}\| \times \cos \alpha_q.$$

Hence,

$$\alpha_q = \arccos \left(\frac{(b_x - a_x)(q_x - b_x) + (b_y - a_y)(q_y - b_y)}{\sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}\sqrt{(q_x - b_x)^2 + (q_y - b_y)^2}} \right).$$

Clearly, computing these angles explicitly is asking for (numerical) trouble.

Can we find the smallest angle α_q *without* using the arccosine and square root operations? Here is the basic observation: To minimize angles, it suffices to *compare* them. That is, given two points q and r, it suffices if we can determine whether $\alpha_q < \alpha_r$, $\alpha_q > \alpha_r$, or $\alpha_q = \alpha_r$. This can be done *without* explicitly knowing these two angles. Consider the following determinant:

$$\Delta(b, q, r) := \begin{vmatrix} b_x & b_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} q_x - b_x & q_y - b_y \\ r_x - b_x & r_y - b_y \end{vmatrix}.$$

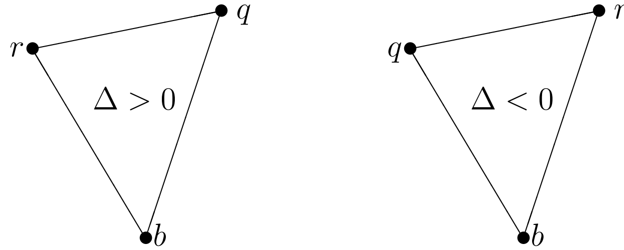


Figure 5: Δ *is twice the signed area of triangle* bqr.

We know from analytic geometry (or linear algebra) that $\Delta(b, q, r)$ is equal to twice the *signed* area of the triangle bqr, where the sign is

$$\begin{array}{rl} \text{positive} & \text{if} \quad \alpha_q < \alpha_r, \\ \text{negative} & \text{if} \quad \alpha_q > \alpha_r, \\ 0 & \text{if} \quad \alpha_q = \alpha_r. \end{array}$$

(See Figure 5.) Therefore, we can compare the angles α_q and α_r by computing the sign of

$$\Delta(b, q, r) = (q_x - b_x)(r_y - b_y) - (q_y - b_y)(r_x - b_x).$$

This only takes five subtractions and two multiplications. It is certainly much simpler than using the arccosine and square root operations.

Assume there is more than one point whose α-angle is minimum. Then we want to select that point among them whose distance to b is maximum. Of course, for each point q with minimum α-angle, we can compute the distance

$$d(b, q) = \sqrt{(b_x - q_x)^2 + (b_y - q_y)^2},$$

and then take that point q for which $d(b, q)$ is maximum. However, we do not have to take square roots: we only have to *compare* the distances $d(b, q)$ and $d(b, r)$. Clearly,

$$d(b, q) < d(b, r) \Longleftrightarrow (b_x - q_x)^2 + (b_y - q_y)^2 < (b_x - r_x)^2 + (b_y - r_y)^2.$$

3 Graham's Scan

The second convex hull algorithm we consider was invented by Graham in 1972. In fact, this was one of the first published computational geometry

algorithms. We give a variant of this algorithm, which is due to Andrew (1979).

Let S be a set of n points in the plane. Let a and b be the leftmost and rightmost points of S, respectively. (If there is more than one leftmost (resp. rightmost) point, then we take for a (resp. b) the lowest (resp. highest) point among the leftmost (resp. rightmost) points. Hence, a and b are the lexicographically smallest and largest elements of S, respectively.) Note that a and b are convex hull vertices.

Let $(p_1, p_2, \ldots, p_h)$ be the vertices of $CH(S)$, where $p_1 = a$. Let r be the index such that $p_r = b$. The line through a and b partitions the convex hull of S into two polygonal chains $(p_1, p_2, \ldots, p_r)$ and $(p_r, p_{r+1}, \ldots, p_h, p_1)$. These chains are called the *lower hull* and *upper hull* of S. See Figure 6.

Clearly, to construct the convex hull of S, it suffices to construct the lower and upper hulls. We will show how to compute the upper hull. The lower hull can be computed in a symmetric way.

All points of $S \setminus \{a, b\}$ that are on or below the line through a and b do not contribute to the upper hull. Therefore, we discard these points. For simplicity, we denote the resulting set by S again. Also, we denote its cardinality by n. If $n = 2$, then there is nothing to do: The upper hull consists of the single edge (a, b).

Assume from now on that $n \geq 3$. Note that all points of $S \setminus \{a, b\}$ are strictly above the line through a and b.

In the algorithm, we will use the following primitive operation: Let x, y, and z be three distinct points. We say that (x, y, z) is a *right turn*, if z is to the right of the directed line segment from x to y. If z is to the left of this segment, then we call (x, y, z) a *left turn*. We saw in Section 2, that

$$(x, y, z) \text{ is a} \begin{cases} \text{right turn} & \text{if } \Delta(x, y, z) < 0, \\ \text{left turn} & \text{if } \Delta(x, y, z) > 0. \end{cases}$$

If $\Delta(x, y, z) = 0$, then the points x, y, and z are collinear.

Here is the algorithm that constructs the upper hull of S. First, we sort the points of S lexicographically. Let

$$a = q_1, q_2, q_3, \ldots, q_n = b$$

be the sorted sequence. Graham's Scan computes the upper hull by considering the points $q_2, q_3, \ldots, q_n$ one after another. It uses a stack $x_0, x_1, \ldots, x_t$ to store points that *may* be upper hull vertices. Point x_t is on the top of the

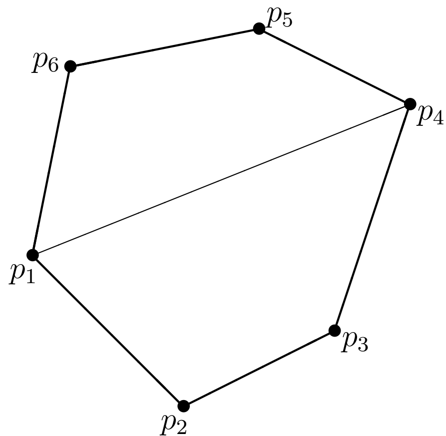


Figure 6: *The line through $a = p_1$ and $b = p_4$ partitions the convex hull into an upper and a lower hull.*

---

stack. At the end of the algorithm, the stack contains the upper hull of S. To be more precise, during the algorithm, we maintain the following

Invariant: The points $x_0, x_1, \ldots, x_t$ on the stack form a subsequence of $q_n, q_1, q_2, \ldots, q_s$, such that

1. $t \geq 2$, $s \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_t = q_s$,
2. $x_1, x_2, \ldots, x_t$ is the upper hull of $q_1, q_2, \ldots, q_s$,
3. the points $x_1, x_2, \ldots, x_t$ are sorted from left to right.

The variables $t, x_0, x_1, \ldots, x_t$ do not occur in the algorithm. They are defined implicitly by the stack, and are only used to prove the correctness of the algorithm.

The complete algorithm is given in Figure 7. We prove the correctness of algorithm *Upper_Hull*. It is clear that the invariant holds immediately after the initialization. Consider one iteration of the outer while-loop, and assume that the invariant holds at the beginning of it.

First note that for all s, $2 \leq s \leq n-1$, (q_{s+1}, x_1, x_0) is a left turn. This implies that x_1 is never popped from the stack. In particular, the stack

```
Algorithm Upper_Hull
(* q_1, q_2, ..., q_n are sorted lexicographically, n ≥ 2, and
   q_2, q_3, ..., q_{n-1} are strictly above the line through q_1 and q_n *)
push(q_n); push(q_1); push(q_2);
s := 2;
(* t = 2, x_0 = q_n, x_1 = q_1, x_t = q_s; the invariant holds *)
while s ≠ n
do α := top element of stack;
   β := second element of stack;
   while (q_{s+1}, α, β) is not a left turn
   do pop the top element from the stack;
      α := β;
      β := second element of stack
   endwhile;
   push(q_{s+1});
   s := s + 1
   (* the invariant holds *)
endwhile;
output the stack
```

Figure 7: *Graham's Scan computes the upper hull of S.*

---

always contains a second element, i.e., point β is always defined. At the end of the iteration, we push q_{s+1} on the stack. Hence, at that moment, the stack contains at least three points, i.e., $t \geq 2$. It easily follows that part 1. of the invariant still holds after the iteration.

Consider the contents $x_0, x_1, \ldots, x_t$ of the stack at the beginning of the iteration. Let i be the index such that $x_t, x_{t-1}, \ldots, x_{i+1}$ are popped during this iteration. Hence, immediately before we increment s, the stack contains the points $x_0, x_1, \ldots, x_i, q_{s+1}$. Note that $1 \leq i \leq t$. Moreover, $x_{i+1}, x_{i+2}, \ldots, x_t$ all lie on or to the right of the line segment from x_i to q_{s+1}. See Figure 8.

Since (q_{s+1}, x_i, x_{i-1}) is a left turn, it follows that $x_1, \ldots, x_i, q_{s+1}$ is the upper hull of $q_1, q_2, \ldots, q_{s+1}$. Immediately before we increment s, the stack just contains the points $x_0, x_1, \ldots, x_i, q_{s+1}$. Therefore, parts 2. and 3. of the invariant hold after we have incremented s.

We have shown that the invariant is correctly maintained during the outer

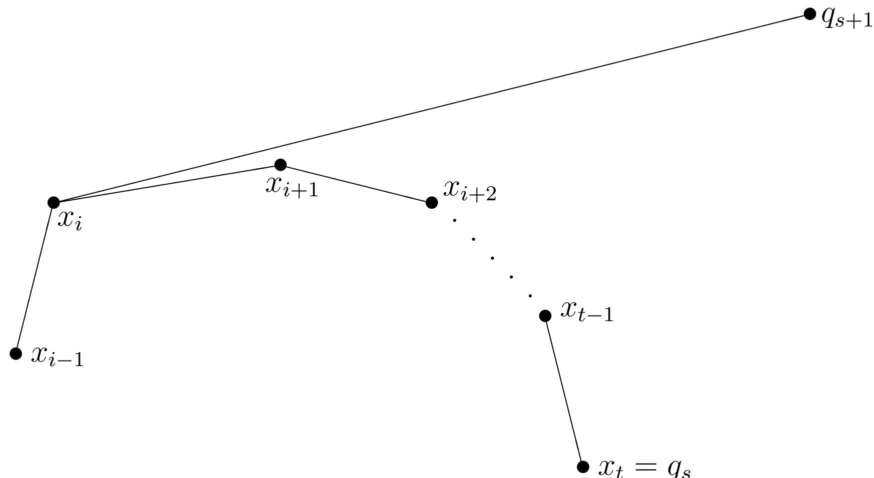


Figure 8: *Illustrating one iteration of algorithm Upper_Hull.*

---

while-loop. Afterwards, we have $s = n$. The invariant implies that

1. $x_1, x_2, \ldots, x_t$ is the upper hull of $q_1, q_2, \ldots, q_n$,
2. the points $x_1, x_2, \ldots, x_t$ are sorted from left to right.

Hence, if we output the stack, starting at its top, then we get the upper hull of S in counterclockwise order. This completes the correctness proof.

Finally, we analyze the running time. It takes $O(n \log n)$ time to sort the points of S lexicographically. Consider algorithm *Upper_Hull*. The initialization takes $O(1)$ time. The outer while-loop makes $n - 2$ iterations. Note that one iteration may take a lot of time, because many elements may be popped.

How do we analyze the total running time of the outer while-loop? It is proportional to the total number of pops and pushes. During one iteration, we push exactly one point. Hence, during the outer while-loop, there are $n-2$ pushes, each one taking $O(1)$ time. To count the total number of pops, we observe that

1. each point is pushed only once, and
2. if a point is popped, then it must have been pushed on the stack before.

It follows that each point is popped at most once from the stack. (If it is popped, then it never reappears on the stack.) Hence, during the outer while-loop, there are at most $n - 2$ pops, each one taking $O(1)$ time.

This proves that—after the sorting step—algorithm *Upper_Hull* takes $O(n)$ time. We have proved the following result.

Theorem 2 *Let $S = \{q_1, q_2, \ldots, q_n\}$ be a set of n points in the plane, that are sorted in lexicographical order. Assume that the points $q_2, q_3, \ldots, q_{n-1}$ are strictly above the line through q_1 and q_n. Then the upper hull of S can be computed in $O(n)$ time.*

Corollary 1 *Let S be an arbitrary set of n points in the plane. The convex hull of S can be computed in $O(n \log n)$ time. If the points of S are sorted lexicographically, then the convex hull of S can be computed in $O(n)$ time.*

4 Computing the convex hull of two convex polygons

Consider the following

Problem 1 Let $P = (p_1, p_2, \ldots, p_m)$ and $Q = (q_1, q_2, \ldots, q_n)$ be two convex polygons. Compute the convex hull of their union.

See Figure 9. It is clear that the convex hull of the union of P and Q is the convex hull of the points $p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_n$.

Let $N := m + n$. Of course, we can solve this problem in $O(N \log N)$ time, using the algorithm of the previous section. Can we use the fact that P and Q are convex polygons to design a faster algorithm? In this section, we show that the problem can be solved in $O(N)$ time. Here are the basic observations:

1. If the entire sequence of vertices of P and Q is sorted lexicographically, then, by Corollary 1, we can compute their convex hull in $O(N)$ time.

2. Since $p_1, p_2, \ldots, p_m$ are the vertices of a convex polygon, we can sort them lexicographically in $O(m)$ time. Similarly, we can sort the vertices of Q in $O(n)$ time.

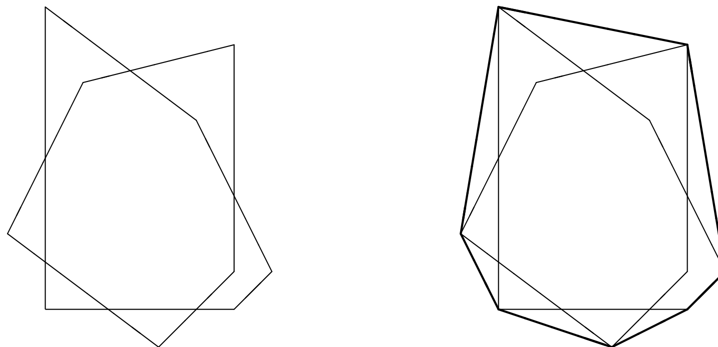


Figure 9: *Two convex polygons and their convex hull.*

The details are as follows. Assume w.l.o.g. that p_1 is the leftmost point of P. Let r be the index such that p_r is the rightmost point. Then P consists of a lower chain $p_1, p_2, \ldots, p_r$, and an upper chain $p_r, p_{r+1}, \ldots, p_m, p_1$.

Since P is convex, the points $p_1, p_2, \ldots, p_r$ are sorted already. Also, the inverse upper chain, i.e., the sequence $p_m, p_{m-1}, \ldots, p_{r+1}$ is sorted already. We merge these sorted sequences, in $O(m)$ time. This gives the vertices of P, sorted in lexicographical order.

Similarly, in $O(n)$ time, we obtain the vertices of Q in lexicographical order. Then these two sorted lists are merged in $O(m+n) = O(N)$ time. Finally, we construct in $O(N)$ time the convex hull of these N points, using the algorithm of the previous section. This proves:

Theorem 3 *Let P and Q be two convex polygons having m and n vertices, respectively. The convex hull of their union can be computed in $O(m+n)$ time.*

This immediately leads to a divide-and-conquer algorithm for constructing the convex hull of an arbitrary set of points. See Figure 10.

It is clear that this algorithm correctly computes the convex hull of S. Let $T(n)$ denote the worst-case running time on an input set of size n. Assuming that n is a power of two, we get the following recurrence relation, for some constant c:

$$T(n) \leq \begin{cases} c & \text{if } n = 1, \\ cn + 2\,T(n/2) & \text{if } n > 1. \end{cases}$$

```
Algorithm Div_Con(S, n)
(* S is a set of n points in the plane *)
if n = 1
then output the only point of S
else m := ⌊n/2⌋;
      partition S arbitrarily into subsets S_1 and S_2
      such that |S_1| = m and |S_2| = n − m;
      P := Div_Con(S_1, m);
      Q := Div_Con(S_2, n − m);
      compute and output the convex hull of P and Q
endif
```

Figure 10: *A divide-and-conquer algorithm that computes the convex hull of a set of points.*

This recurrence is solved by unfolding it:

$$\begin{aligned} T(n) &\leq cn + 2\,T(n/2) \\ &\leq cn + 2\,(c \cdot n/2 + 2\,T(n/4)) \\ &= 2cn + 2^2\,T(n/2^2) \\ &\leq 2cn + 2^2\left(c \cdot n/2^2 + 2\,T(n/2^3)\right) \\ &= 3cn + 2^3\,T(n/2^3) \\ &\ \ \vdots \\ &\leq icn + 2^i\,T(n/2^i). \end{aligned}$$

For $i = \log n$, we get

$$T(n) \leq cn \log n + n\,T(1) = O(n \log n).$$

Hence, we have another $O(n \log n)$-time algorithm that computes the convex hull of a set of n points in the plane.

Exercise 4 For arbitrary values of n, the running time $T(n)$ satisfies

$$T(n) \leq \begin{cases} c & \text{if } n = 1, \\ cn + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & \text{if } n > 1. \end{cases}$$

Prove, e.g. by induction, that $T(n) = O(n \log n)$.

5 A variant of Graham's Scan and triangulations

In this section, we give a variant of Graham's algorithm. As we will see, the new algorithm computes more information than just the convex hull. Instead of computing the upper and lower hulls separately, it computes the entire convex hull in one pass along the points.

Let S be a set of n points in the plane, sorted in lexicographical order. The algorithm constructs the convex hull incrementally, from left to right. To be more precise, let $p_1, p_2, \ldots, p_n$ be the sorted sequence of points. We visit the points one after another.

Invariant: After we have visited the i-th point, we have constructed the convex hull of $\{p_1, p_2, \ldots, p_i\}$.

What happens during one iteration? Assume we have the convex hull of $\{p_1, p_2, \ldots, p_{i-1}\}$. Then we have to "add" point p_i, i.e., we have to compute the convex hull of

$$\{p_1, p_2, \ldots, p_i\} = \{p_1, p_2, \ldots, p_{i-1}\} \cup \{p_i\}.$$

This basically means that we have to find the two *tangents* from p_i to the current convex hull, see Figure 11. (Note that p_i is outside the current hull.)

How do we find these tangents? Assume that the vertices of the current hull are stored in a doubly-linked list, sorted in counterclockwise order. We denote the successor and predecessor of an element p in this list by $succ(p)$ and $pred(p)$, respectively. The upper tangent is found as follows:

> $\alpha := p_{i-1}$; $\beta := succ(\alpha)$;
> **while** (p_i, α, β) is not a left turn
> **do** $\alpha := \beta$; $\beta := succ(\alpha)$
> **endwhile**

Similarly, the lower tangent is found by the following procedure:

> $\gamma := p_{i-1}$; $\delta := pred(\gamma)$;
> **while** (p_i, γ, δ) is not a right turn
> **do** $\gamma := \delta$; $\delta := pred(\gamma)$
> **endwhile**

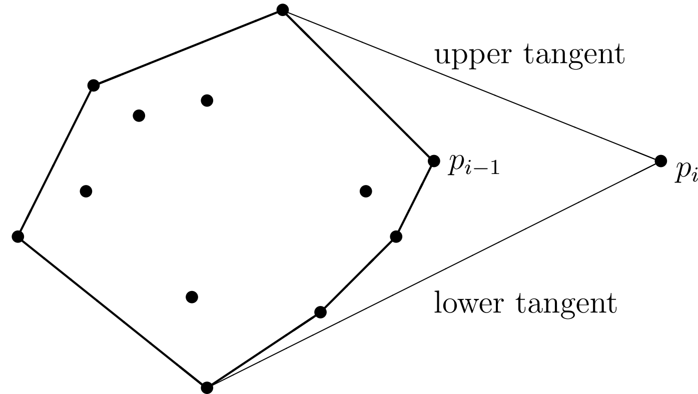


Figure 11: *We have to find the two tangents from p_i to the current convex hull.*

Consider the values of α and γ after these two procedures. The convex hull of $\{p_1, p_2, \ldots, p_i\}$ is obtained by replacing the sequence

$$succ(\gamma), succ(succ(\gamma)), \ldots, pred(pred(\alpha)), pred(\alpha)$$

by point p_i. This is done as follows, see Figure 12:

$$succ(\gamma) := p_i;\ pred(p_i) := \gamma;\ succ(p_i) := \alpha;\ pred(\alpha) := p_i;$$

Exercise 5 Convince yourself that in this way, we get an $O(n)$-time convex hull algorithm. (Recall that we assume that the points are sorted already.)

By changing this algorithm slightly, it computes more information than just the convex hull. Again, we maintain the convex hull of $\{p_1, p_2, \ldots, p_{i-1}\}$ in a doubly-linked list, which we denote by L. Moreover, we maintain an initially empty graph G having the points $p_1, p_2, \ldots, p_n$ as its vertices. The new algorithm is given in Figure 13.

The only difference with the previous algorithm is the graph G. We not only replace the sequence

$$succ(\gamma), succ(succ(\gamma)), \ldots, pred(pred(\alpha)), pred(\alpha)$$

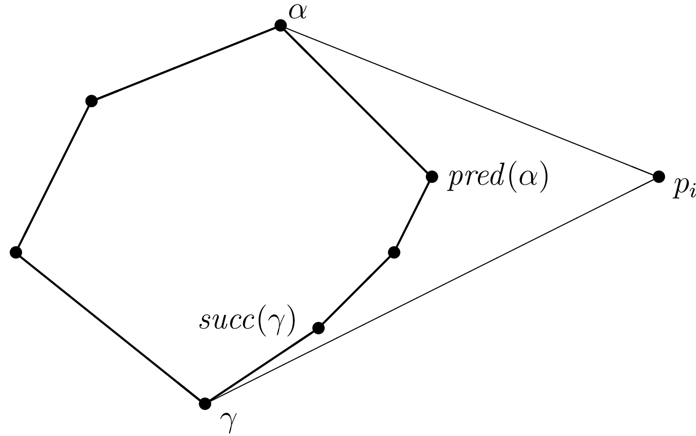


Figure 12: *To update the current convex hull, we replace the edges between γ and α by the two tangents from p_i.*

---

by p_i, we also add the edges

$$(\gamma, p_i), (succ(\gamma), p_i), \ldots, (pred(\alpha), p_i), (\alpha, p_i)$$

to G. In Figure 14, you can see how this graph looks like at the end of the algorithm.

Exercise 6 Prove that this algorithm still runs in $O(n)$ time.

What kind of graph is G? Consider what happens during the i-th iteration of the outer while-loop: we add edges (γ, p_i), $(succ(\gamma), p_i)$, $\ldots$, (α, p_i) to G. It is clear that these new edges do not intersect each other. Also, they do not intersect edges that were added during previous iterations. Hence, G is a planar graph. It is easy to see that each bounded face of G is a triangle. Therefore, the graph G is called a *triangulation* of the point set S.

This observation leads to another proof of the fact that the running time of algorithm *Triangulate* is linear: Consider one iteration of the outer while-loop. It consists of three inner while-loops. During the third inner loop, we visit the same points as during the other two inner loops. Therefore, the time for one iteration of the outer loop is proportional to the time for the third inner loop. The latter is proportional to the number of edges that are added to G.

Algorithm *Triangulate*
(∗ $p_1, p_2, \ldots, p_n$ are sorted lexicographically ∗)
if (p_1, p_2, p_3) is a left turn
then $L := \langle p_1, p_2, p_3 \rangle$
else if (p_1, p_2, p_3) is a right turn
    then $L := \langle p_1, p_3, p_2 \rangle$
    else $L := \langle p_1, p_3 \rangle$
    endif
endif;
add the edges (p_1, p_2), (p_2, p_3), and (p_1, p_3) to G;
$i := 4$;
while $i \leq n$
do $\alpha := p_{i-1}$; $\beta := succ(\alpha)$;
   while (p_i, α, β) is not a left turn
   do $\alpha := \beta$; $\beta := succ(\alpha)$
   endwhile;
   $\gamma := p_{i-1}$; $\delta := pred(\gamma)$;
   while (p_i, γ, δ) is not a right turn
   do $\gamma := \delta$; $\delta := pred(\gamma)$
   endwhile;
   $x := \gamma$;
   while $x \neq succ(\alpha)$
   do add edge (x, p_i) to G;
      $x := succ(x)$
   endwhile;
   $succ(\gamma) := p_i$; $pred(p_i) := \gamma$;
   $succ(p_i) := \alpha$; $pred(\alpha) := p_i$;
   $i := i + 1$
endwhile

Figure 13: *Computing the convex hull and a triangulation.*

Hence, the entire running time of the algorithm is proportional to the total number of edges that are added to G. Since G is a planar graph on n vertices, we know from Euler's formula that it has at most $3n - 6$ edges. This proves that the algorithm runs in $O(n)$ time.

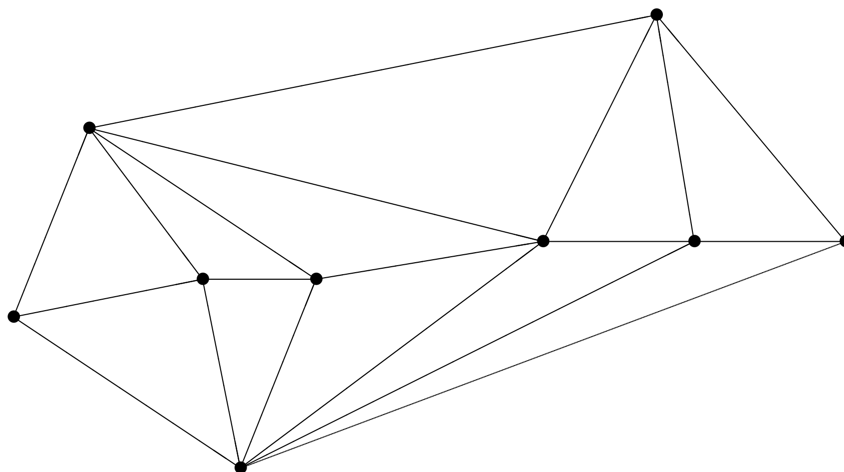


Figure 14: *The output of algorithm Triangulate.*

Theorem 4 *Let S be a set of n points in the plane, sorted in lexicographical order. In $O(n)$ time, we can compute a triangulation of S.*

Exercise 7 Let S be a set of n points in the plane, and let h be the number of vertices of its convex hull. Prove that any triangulation of S contains exactly $3n - h - 3$ edges and $2n - h - 2$ bounded faces.

6 An optimal output-sensitive convex hull algorithm

Until now, we have seen two types of algorithms that solve the convex hull problem:

1. Graham's Scan and a variation of it, and a divide-and-conquer algorithm. These algorithms have a running time of $O(n \log n)$.

2. Jarvis' March. If h denotes the number of vertices of the convex hull, then this algorithm has a running time of $O(nh)$. Such an algorithm, whose running time depends both on n (the input size), and h (the output size), is called *output-sensitive.*

There are two remarks to be made. First, Graham's Scan is optimal, in the sense that for any convex hull algorithm $\mathcal{A}$, there is a set S of n points in the plane, such that $\mathcal{A}$ needs $\Omega(n \log n)$ time to compute $CH(S)$.

Second, assume that h, the number of vertices of $CH(S)$, is much smaller than $\log n$. Then, Jarvis' March is faster than Graham's Scan. E.g., if h is a constant, then Jarvis' March has linear running time, whereas Graham's algorithm takes $\Theta(n \log n)$ time. On the other hand, if h is much larger than $\log n$, then Graham's Scan is faster. E.g., if $h = n$, then Jarvis' algorithm takes quadratic time, whereas Graham's algorithm takes only $\Theta(n \log n)$ time.

This leads to the following problem: Is there a convex hull algorithm that is at least as fast as both Jarvis' March and Graham's Scan, on any input?

Exercise 8 Show that this problem has a positive answer, by giving a convex hull algorithm with running time

$$O(\min(n \log n, nh)).$$

In this section, we will show that there is an even faster algorithm. That is, we will prove the following result.

Theorem 5 *Let S be a set of n points in the plane, and let h be the number of vertices of its convex hull. There is an algorithm that computes the convex hull of S in $O(n \log h)$ time.*

This theorem was proved in 1982 by David Kirkpatrick and Raimund Seidel. They gave a quite complicated algorithm. We will present an *extremely simple* $O(n \log h)$-time algorithm that was discovered in 1994 by Timothy Chan. At that time, he was a 19-year old PhD student at the University of British Columbia in Vancouver. Surprisingly, Chan's algorithm only uses techniques that have been known since the beginning of the eighties. Basically, his algorithm is a clever implementation of Jarvis' March.

Before we can give Chan's algorithm, we need some results on representing convex polygons.

6.1 The hierarchical representation of convex polygons

We consider the following problem. Let S be a set of n points in the plane. We want to store these points in a data structure, such that so-called *extremal queries* can be answered. In such a query, we are given a point q in the plane,

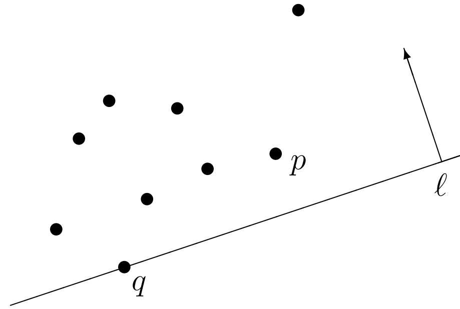


Figure 15: *When we rotate ℓ around q, point p is hit first.*

---

and a line ℓ through q, such that all points of S are on or above ℓ. The answer to the query is a point of S that is hit first when we rotate ℓ around q in counterclockwise order. See Figure 15.

It is clear that a point p can be the answer to an extremal query only if it is a vertex of the convex hull of S. Let $P = (p_0, p_1, \ldots, p_{h-1})$ be the vertices of $CH(S)$, enumerated in counterclockwise order. Then we only have to consider the elements of P. We represent these points using the *hierarchical representation* of P. (Invented by Chazelle and Dobkin (1980), and Dobkin and Kirkpatrick (1983).) This representation is obtained as follows.

Let $P_0 := P$. For $i \geq 1$, let P_i be the sequence obtained from P_{i-1}, by deleting every second point. Let k be the index such that P_k contains exactly two elements. Then the sequence of sequences

$$H(P) = (P_0, P_1, \ldots, P_k)$$

is called the hierarchical representation of P. See Figure 16. For $0 \leq i \leq k$, let $h_i := |P_i|$, the number of elements of P_i.

Exercise 9 Prove that

1. $h_i \leq \frac{1}{2}(h_{i-1} + 1)$,
2. $h_i \leq (h + 2^i - 1)/2^i$,
3. $h_i < 1 + h/2^i$,
4. $k < \log h$.

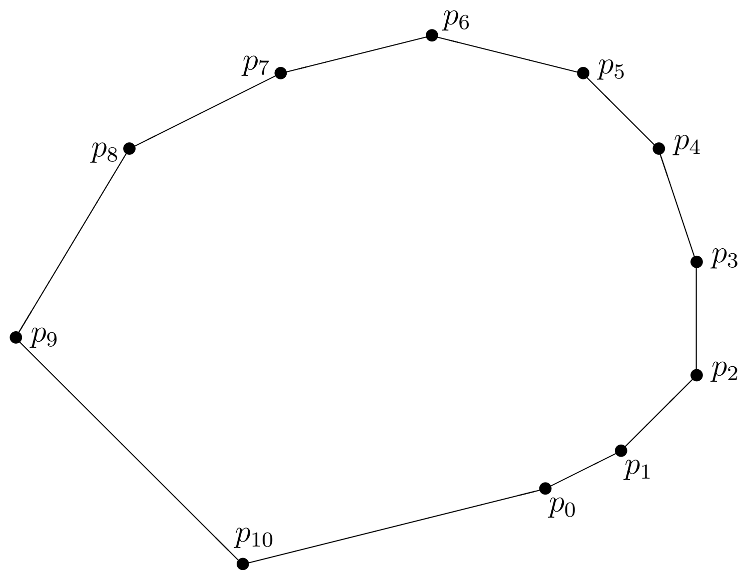


Figure 16: *The hierarchical representation of the convex polygon P consists of $P_0 = P$, $P_1 = (p_0, p_2, p_4, p_6, p_8, p_{10})$, $P_2 = (p_0, p_4, p_8)$, and $P_3 = (p_0, p_8)$.*

Exercise 10 Prove that each P_i is a convex polygon.

We claim that this hierarchical representation can be used to answer extremal queries. Indeed, let ℓ be a line such that all points of S are on or above it, and let q be an arbitrary point on ℓ. Let p be the point of S that is hit first when we rotate ℓ around q in counterclockwise order. Our goal is to find p. Note that $p \in P_0 = P$.

For $0 \leq i \leq k$, let q_i be the point of P_i that is hit first when rotating ℓ around q. We compute point p as follows.

1. First, we find q_k.
2. Then, for $i = k, k-1, \ldots, 1$, assuming that we know q_i, we compute q_{i-1}.
3. At this moment, we know q_0. Since $q_0 = p$, we report this point.

How do we implement these steps? Since the sequence P_k contains only two elements, point q_k can easily be computed in constant time. Let $1 \leq$

$i \le k$, and assume that we have computed point $q_i \in P_i$ already. How do we compute q_{i-1}? The following lemma gives the answer.

Lemma 1 *Let a and b be the predecessor and successor of q_i in P_{i-1}. Then $q_{i-1} \in \{a, q_i, b\}$.*

Exercise 11 Prove Lemma 1. (*Hint:* Looking at some examples will give you an idea why this lemma is true.)

Lemma 1 implies that given q_i, we can compute q_{i-1} in constant time. Hence, the entire algorithm to find $p = q_0$ takes time $O(k) = O(\log h) = O(\log n)$. In order to implement the query algorithm, we have to specify how to store the hierarchical representation. The straightforward solution is to store each sequence P_i as a doubly-linked circularly ordered list, and give each element in P_i a pointer to its occurrence in P_{i-1}.

Exercise 12 Convince yourself that by storing the hierarchical representation this way, the query algorithm can be implemented such that it runs in $O(\log n)$ time.

How much space do we need to store the data structure? For each sequence P_i, we need $O(|P_i|) = O(h_i)$ space. Hence, the amount of space for the entire data structure is

$$O\left(\sum_{i=0}^{k} h_i\right) = O\left(\sum_{i=0}^{k}(1 + h/2^i)\right) = O(k+h) = O(n).$$

The data structure can be built as follows.

1. Compute the convex hull of the set S using, e.g., Graham's Scan. This gives us the sequence P_0.
2. For $i = 1, 2, \ldots$, construct the sequence P_i from P_{i-1}, by deleting every second point, and establish the pointers from P_i to P_{i-1}.

The first step of this algorithm takes $O(n \log n)$ time. The second step takes time $O(\sum_{i=0}^{k} h_i) = O(n)$. We have proved the following result.

Theorem 6 *Let S be a set of n points in the plane. The hierarchical representation of S has size $O(n)$, and can be built in $O(n \log n)$ time. It can be used to solve extremal queries in $O(\log n)$ time.*

Exercise 13 We stored the hierarchical representation using a sequence of lists. Show that in fact, it can be stored in one single array.

6.2 Chan's convex hull algorithm

Let S be a set of n points in the plane, and let h be the number of vertices of the convex hull $CH(S)$ of S. We show how to compute this convex hull in $O(n \log h)$ time.

First assume that we know the value of h. (Of course, h is not known in advance! But, let us assume for the moment that someone has told us this value.) Here is the algorithm.

Step 1: Partition the set S (arbitrarily) into subsets $S_1, S_2, \ldots S_{\lceil n/h \rceil}$, each of size h, except possibly the last one, which may be smaller.

Step 2: For each i, $1 \leq i \leq \lceil n/h \rceil$, construct the data structure of Theorem 6 for the set S_i.

Step 3: Now we compute the convex hull of the entire set S using the gift wrapping technique. Let p_0 be the point of S having minimum y-coordinate. (If there are several points with minimum y-coordinate, then we take the leftmost of these points.) Note that p_0 is a vertex of $CH(S)$.

Let p_1 be the point of S such that $p_0 p_1$ is a "counterclockwise" edge of $CH(S)$. We find p_1 as follows.

Let ℓ be the horizontal line through p_0. Note that all points of S are on or above ℓ. For each i, $1 \leq i \leq \lceil n/h \rceil$, query the data structure for S_i, i.e., find the point $q_i \in S_i$ that is hit first when rotating ℓ around p_0 in counterclockwise order. Then, by a linear search, find the point q_j among $q_1, q_2, \ldots, q_{\lceil n/h \rceil}$ for which the angle between ℓ and $p_0 q_j$ is minimum. (If there are several points for which this angle is minimum, then we take the one having maximum distance from p_0.) It is easy to see that $p_1 = q_j$.

Given p_1, we find the next convex hull vertex p_2 in a similar way: Let ℓ' be the line through p_0 and p_1. Again, all points of S are on one side of ℓ'. We perform an extremal query in each subset S_i, $1 \leq i \leq \lceil n/h \rceil$, by rotating ℓ' in counterclockwise order around the point p_1. Then p_2 is the point x among the $\lceil n/h \rceil$ answers for which the angle between ℓ' and $p_1 x$ is minimum. Of course, we keep on making these gift wrapping steps, until we are back at our starting point p_0.

This concludes the description of the algorithm. It is easy to see that this algorithm correctly computes the convex hull of S. What is the running time? Step 1 takes $O(n)$ time. By Theorem 6, the data structure for one subset S_i can be built in $O(|S_i| \log |S_i|) = O(h \log h)$ time. Hence, Step 2

takes total time

$$O\left(\frac{n}{h}\, h \log h\right) = O(n \log h).$$

Let us consider Step 3. The lowest point p_0 can be found in $O(n)$ time. To find p_1, we perform one extremal query in each subset S_i. Each such query takes time $O(\log |S_i|) = O(\log h)$. Hence, all queries together take $O((n/h) \log h)$ time. They give us $\lceil n/h \rceil$ candidates for the point p_1. In $O(n/h)$ time, p_1 is selected from these candidates. Hence, given p_0, we find p_1 in $O((n/h) \log h)$ time. In general, given the k-th convex hull vertex p_k, we find the next convex hull vertex p_{k+1} in $O((n/h) \log h)$ time. Since there are h convex hull vertices, the total time for Step 3 is

$$O\left(n + h\, \frac{n}{h} \, \log h\right) = O(n \log h).$$

Hence, the entire algorithm for constructing the convex hull of S takes $O(n \log h)$ time.

Note that this only works if the value of h is known in advance. What do we do if h is not known? The trick is to "guess" h in a clever way.

Assume H is our current guess value. Then we run the above algorithm with h replaced by H. Steps 1 and 2 do not cause any problems; they are completed in $O(n \log H)$ time. In Step 3, two things can happen:

1. We are back at the starting point p_0 within H gift wrapping steps. This happens if and only if $h \leq H$. In this case, the convex hull of S is constructed in Step 3, in time

   $$O\left(n + h\, \frac{n}{H} \, \log H\right) = O(n \log H),$$

   and we are done.

2. After H gift wrapping steps, we are not back at p_0. This happens if and only if $h > H$, i.e., our guess value for the number of convex hull vertices is too small. In this case, we stop the algorithm after these H gift wrapping steps. Note that in this case, we also spend $O(n \log H)$ time, but we still do not have the convex hull of S. Since our guess value was too small, we try a larger value for H, and run the entire algorithm again.

Which guess values do we take? We start with $H = 4$, or any other small integer greater than one. Each time we discover that $H < h$, we set $H := H^2$.

Since the value of H strictly increases, it will become greater than or equal to h. When this happens for the first time, we will compute the complete convex hull of S, and we are done.

Let H_f be the final value of H. That is, when running the algorithm with guess value H_f, we complete the convex hull construction within H_f gift wrapping steps. Then we know that $H_f \geq h$. The previous guess value, which is equal to $\sqrt{H_f}$, was less than h. Hence, $H_f < h^2$.

Now we can bound the running time of the complete algorithm. We know that for each guess value H, we spend at most $cn \log H$ time, for some constant c. Hence, for the final guess value H_f, we spend at most $cn \log H_f$ time. For the second last guess value, we spend at most

$$cn \log \sqrt{H_f} = \frac{1}{2} cn \log H_f$$

time. In general, for the i-th last guess value, we spend at most

$$cn \log H_f^{(1/2)^{i-1}} = \left(\frac{1}{2}\right)^{i-1} cn \log H_f$$

time. Therefore, the total running time of our algorithm is bounded from above by

$$\begin{aligned} \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^{i-1} cn \log H_f &= 2cn \log H_f \\ &< 2cn \log h^2 \\ &= 4cn \log h \\ &= O(n \log h). \end{aligned}$$

This proves Theorem 5.

7 Duality and computing the intersection of halfplanes

In the previous sections, we have seen several convex hull algorithms. In this section, we will see that we can use these algorithms to solve a seemingly unrelated problem.

A *halfplane* is the set of all points in $\mathbb{R}^2$ that are on one side of a given line. This line itself also belongs to the halfplane. The problem we will solve is the following: We are given a set of n halfplanes, and want to compute their intersection. It is easy to see that the intersection of n halfplanes is a (possibly unbounded) convex polygon. We say that we have solved the problem, if we have the vertices of this polygon, sorted in counterclockwise order.

Exercise 14 Prove that the intersection of n halfplanes has at most n vertices.

In order to show that this problem can be solved using any convex hull algorithm, we need a *duality transformation.* This transformation maps points to non-vertical lines, and non-vertical lines to points.

7.1 The duality transformation

Let ℓ be any non-vertical line, having equation $y = ax + b$. The *dual* of ℓ is the point

$$D(\ell) := (a, b) \in \mathbb{R}^2.$$

Conversely, let $p = (p_1, p_2)$ be any point in the plane. The *dual* of p is the non-vertical line

$$D(p) : y = -p_1 x + p_2.$$

Here is the basic property of this transformation.

Lemma 2 *Let $p = (p_1, p_2)$ be a point in the plane, and let $\ell : y = ax + b$ be a non-vertical line.*

1. *p is below ℓ if and only if the point $D(\ell)$ is above the line $D(p)$.*

2. *p is above ℓ if and only if the point $D(\ell)$ is below the line $D(p)$.*

3. *p is on ℓ if and only if the point $D(\ell)$ is on the line $D(p)$.*

Proof. We only prove the first claim. Point p is below ℓ if and only if $p_2 < ap_1 + b$. This inequality is equivalent to $b > -ap_1 + p_2$, which holds if and only if point $D(\ell)$ is above line $D(p)$. ∎

7.2 Applying the duality transformation

If ℓ is a non-vertical line, then ℓ^+ and ℓ^- denote the halfplanes above and below ℓ, respectively.

Consider an input to the halfplane intersection problem. We assume for simplicity that no halfplane is bounded by a vertical line[2]. The input consists of a number of "upper" halfplanes, and a number of "lower" halfplanes. Hence, there is an integer m, $0 \leq m \leq n$, such that we can write the input as

$$\ell_1^+, \ell_2^+, \ldots, \ell_m^+, \ell_{m+1}^-, \ell_{m+2}^-, \ldots, \ell_n^-.$$

Let

$$S^+ := \bigcap_{i=1}^{m} \ell_i^+,$$

and

$$S^- := \bigcap_{i=m+1}^{n} \ell_i^-.$$

It is clear that the intersection of the n halfplanes is equal to the intersection of S^+ and S^-. We will show how to compute S^+. The intersection S^- can be computed in a symmetric way. At the end, we will show how to compute the intersection of S^+ and S^-.

The set S^+ is an unbounded convex polygon. Each edge of this polygon is contained in a line ℓ_j, for some j, $1 \leq j \leq m$. We call a line ℓ_i, where $1 \leq i \leq m$, *redundant*, if it does not contribute to S^+, i.e., there is no edge of S^+ that is contained in ℓ_i. See Figure 17.

We now prove two lemmas which show the relationship between S^+ and the upper hull of the points $D(\ell_i)$, $1 \leq i \leq m$. Only the second lemma is needed to prove the correctness of our algorithm that constructs S^+. We include the first lemma to illustrate the way duality is used.

Lemma 3 *Let $1 \leq i \leq m$. Line ℓ_i is redundant if and only if the point $D(\ell_i)$ is not a vertex of the upper hull of the points $D(\ell_1), D(\ell_2), \ldots, D(\ell_m)$.*

Proof. Assume that ℓ_i is redundant. Then $S^+ \cap \ell_i$ is either empty, or a vertex of S^+. Let v be a vertex of S^+ that is closest to ℓ_i. See Figure 17.

[2] In theory, this is a reasonable assumption, because we can always rotate the coordinate system.

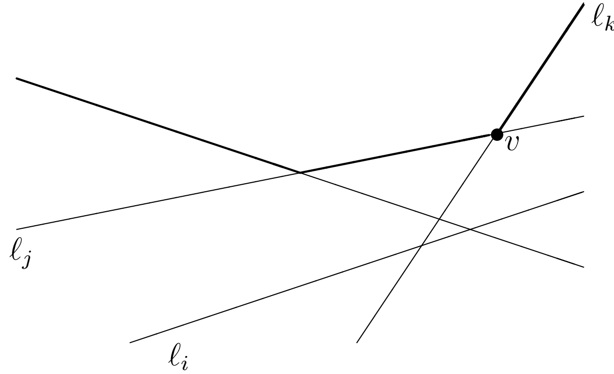
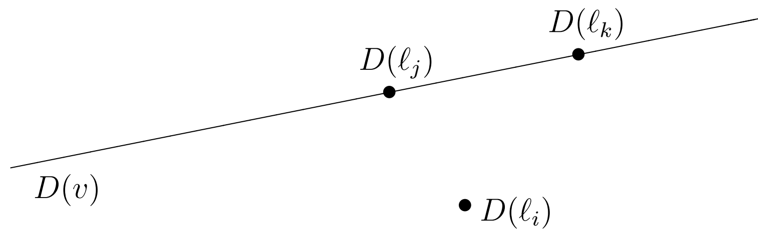


Figure 17: *Illustrating the proof of Lemma 3. Line ℓ_i is redundant because it does not contribute to the intersection S^+.*

There are two indices j and k, $1 \le j < k \le m$, such that ℓ_j and ℓ_k are non-redundant, and v is the intersection between ℓ_j and ℓ_k. Note that ℓ_j and ℓ_k "witness" the redundancy of ℓ_i.

Since v is on or above ℓ_i, we know that the point $D(\ell_i)$ is on or below the line $D(v)$. Observe that $D(v)$ contains the two points $D(\ell_j)$ and $D(\ell_k)$.

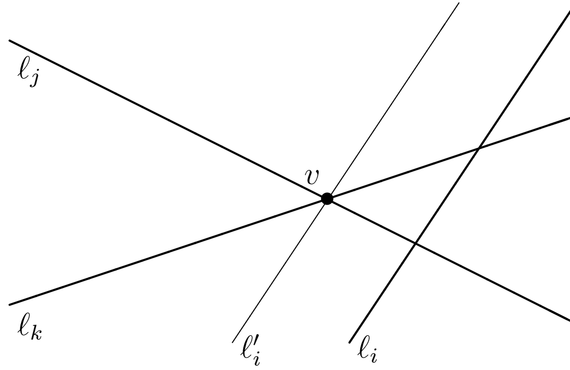
We claim that the x-coordinate of $D(\ell_i)$ is between the x-coordinates of $D(\ell_j)$ and $D(\ell_k)$. This claim will prove that $D(\ell_i)$ is not a vertex of the upper hull of $D(\ell_1), D(\ell_2), \ldots, D(\ell_m)$. See the figure below.



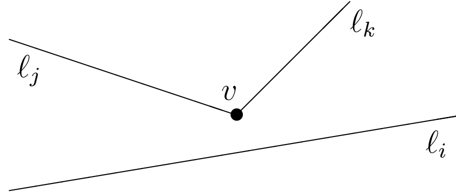
To prove the claim, let $D(\ell_i)$ be the point (a_i, b_i), and similarly, let $D(\ell_j) = (a_j, b_j)$, and $D(\ell_k) = (a_k, b_k)$. Assume w.l.o.g. that $a_j < a_k$. Then we have to show that $a_j \le a_i \le a_k$.

First note that the slope of ℓ_j is smaller than that of ℓ_k. Assume that $a_i > a_k$. Then the line ℓ_i has a larger slope than ℓ_k. Let ℓ_i' be the line through v that is parallel to ℓ_i. (See the figure below.) Recall that v is a vertex of

S^+ that is closest to ℓ_i. Hence, there are no vertices of S^+ between ℓ'_i and ℓ_i. Since S^+ is above ℓ_i, it is in fact above ℓ'_i. But then, line ℓ_k is redundant, which is a contradiction. Hence, $a_i \leq a_k$. In a symmetric way, we can prove that $a_j \leq a_i$. This proves the first part of the lemma.



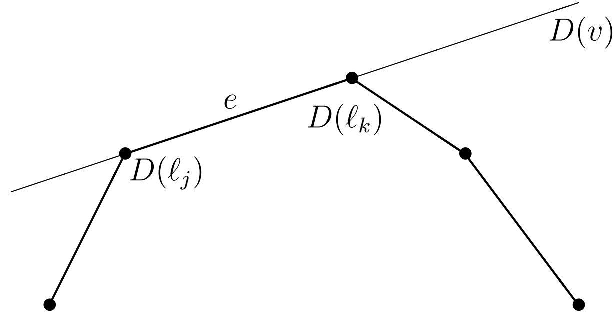
The converse of the proof is similar. Assume that $D(\ell_i)$ is not a vertex of the upper hull of the points $D(\ell_1)$, $D(\ell_2)$, ..., $D(\ell_m)$. Then this upper hull contains two vertices, say $D(\ell_j)$ and $D(\ell_k)$, such that $D(\ell_i)$ is below the edge defined by these two vertices. Let v be the intersection of ℓ_j and ℓ_k. Then v is above ℓ_i and the situation looks as follows:



Since S^+ is above ℓ_j and ℓ_k, it follows that ℓ_i does not contribute to S^+. That is, ℓ_i is redundant. ∎

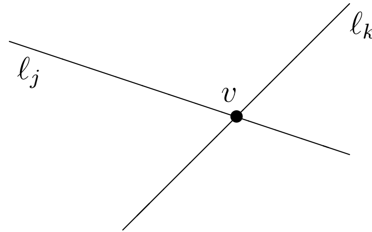
Lemma 4 *Let v be any point in the plane. Then, v is a vertex of S^+ if and only if the line $D(v)$ contains an edge of the upper hull of the points $D(\ell_1), D(\ell_2), \ldots, D(\ell_m)$.*

Proof. Assume that $D(v)$ contains edge e of the upper hull of $D(\ell_1)$, $D(\ell_2)$, ..., $D(\ell_m)$. Let $D(\ell_j)$ and $D(\ell_k)$ be the endpoints of e. Since the line $D(v)$ contains $D(\ell_j)$ and $D(\ell_k)$, we know that v is the intersection of ℓ_j and ℓ_k.



All points $D(\ell_i)$, $1 \leq i \leq m$, are on or below the line $D(v)$. Therefore, v is on or above all lines ℓ_i, $1 \leq i \leq m$. But this means that $v \in S^+$.

Since $S^+ \subseteq \ell_j^+ \cap \ell_k^+$, we conclude that v is a vertex of S^+.



The converse can be proved similarly. ∎

7.3 The algorithm

Now we can give the algorithm that constructs S^+. Recall that the input consists of m non-vertical lines $\ell_1, \ell_2, \ldots, \ell_m$. We want to compute $S^+ = \cap_{i=1}^m \ell_i^+$.

Step 1: Compute the points $p_i := D(\ell_i)$, $1 \leq i \leq m$.

Step 2: Compute the upper hull of the points $p_1, p_2, \ldots, p_m$. Let $e_1, e_2, \ldots, e_h$ be the edges of this upper hull, sorted from left to right.

Step 3: For each i, $1 \leq i \leq h$, compute the line L_i that contains e_i.

Step 4: Output the points $D^{-1}(L_i)$, $1 \leq i \leq h$.

Lemma 4 implies that in Step 4, all vertices of S^+ are reported. The edges $e_1, e_2, \ldots, e_h$ are sorted from left to right. Since these edges are on the upper hull, they are also sorted by slope. Of course, the same holds for the lines

$L_1, L_2, \ldots, L_h$. Then, the definition of the duality transformation D implies that the points $D^{-1}(L_1), D^{-1}(L_2), \ldots, D^{-1}(L_h)$ are also sorted. Hence, the algorithm computes all vertices of S^+, sorted along its boundary. This proves the correctness of the algorithm.

Exercise 15 The algorithm computes the vertices of S^+. How can we find the edges? In particular, how do we find the leftmost and rightmost edges (which are unbounded)?

The running time of the algorithm is easy to estimate. Steps 1, 3 and 4 take $O(m)$ time. If we use Graham's Scan, then Step 2 takes $O(m \log m)$ time. In fact, if the lines $\ell_1, \ell_2, \ldots, \ell_m$ are sorted by slope, then the points $p_1, p_2, \ldots, p_m$ are sorted by x-coordinate, and Step 2 takes only $O(m)$ time. We have proved the following result.

Theorem 7 *Let $\ell_1, \ell_2, \ldots, \ell_m$ be non-vertical lines that are sorted by slope. In $O(m)$ time, we can compute the boundary of the intersection $S^+ = \cap_{i=1}^m \ell_i^+$.*

Recall that we wanted to compute the boundary of

$$S^+ \cap S^- = \left(\bigcap_{i=1}^{m} \ell_i^+ \right) \cap \left(\bigcap_{i=m+1}^{n} \ell_i^- \right).$$

Of course, the boundary of S^- can be computed in a symmetric way: First dualize the lines ℓ_i to points p_i, $m+1 \leq i \leq n$. Then compute the lower hull of these points. Finally, compute the inverse $D^{-1}(L)$ of each line L that contains a lower hull edge.

So it remains to show how the intersection of S^+ and S^- can be computed. Let h^+ and h^- denote the number of vertices of S^+ and S^-, respectively. Note that the edges of S^+ and S^- are sorted from left to right. By walking simultaneously along these edges, we can easily compute the at most two intersection points between the boundaries of S^+ and S^- in time $O(h^+ + h^-) = O(n)$. This solves our problem of computing the intersection of halfplanes.

Theorem 8 *The intersection of n halfplanes can be computed in $O(n \log n)$ time. If the lines that bound the halfplanes are sorted by slope, then the intersection can be computed in $O(n)$ time.*

8 Some final remarks

In these notes, we have only considered convex hull algorithms for points in the plane. It follows from Euler's theorem that the convex hull of n points in $\mathbb{R}^3$ has size $O(n)$, i.e., the total number of vertices, edges and facets is $O(n)$. Preparata and Hong proved in 1977 that the convex hull can be computed in $O(n \log n)$ time. There are more practical algorithms for the three-dimensional case. See the latest books on computational geometry. Chan's algorithm can be generalized to three dimensions. If h is the number of hull vertices, then the entire convex hull can be computed in $O(n \log h)$ time.

In dimensions larger than three, the situation becomes more complex. Let S be a set of n points in $\mathbb{R}^d$, where d is a constant. The total size (i.e., the total number of vertices, edges, 2-dimensional facets, ..., $(d-1)$-dimensional facets) is $\Theta(n^{\lfloor d/2 \rfloor})$ in the worst-case. In 1981, Seidel showed that the convex hull can be computed in $O(n^{\lceil d/2 \rceil})$ time. Hence, for *odd* dimensions, there is one factor n "too much", whereas for *even* dimensions, this is optimal[3]. In 1986, Seidel also gave an algorithm that computes the convex hull in $O(n^{\lfloor d/2 \rfloor} \log n)$ time. For quite some time, these were the best general results for the convex hull problem. In 1991, Chazelle solved the problem optimally: he showed how to compute the convex hull in $O(n^{\lfloor d/2 \rfloor})$ time.

[3]this sounds odd, doesn't it?