Computational Geometry

Professor Dr.Thomas Ottmann
Albert Ludwigs University
Freiburg

Lecture 1: Introduction

- History: Proof-based, agorithmic, axiomatic geometry, computational geometry today
- Problems and applications
- An example: Computing the convex hull:
  1. the "naive approach"
  2. Graham's Scan
  3. Lower bound
- Design, analysis, and implementation of geometrical algorithms

Ancient example of proof-based geometry

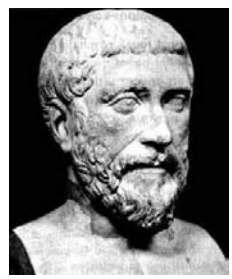
Pythagoras´s Theorem (562 - 475 BC):

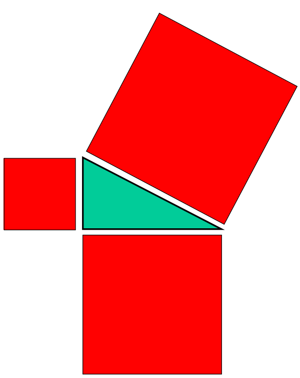
The sum of the squares of the sides of a right triangle is equal to the square of the hypotenuse.

Already known to the Babylonians and Egyptians as experimental fact.
Pythagorean innovation: A proof, independent of experimental numerical verification.

Pythagoras

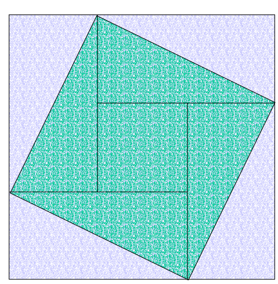
Born: about 562 BC in Samos
Died: about 475 BC





Pythagoras´s theorem

Proof of Pythagoras´s Theorem



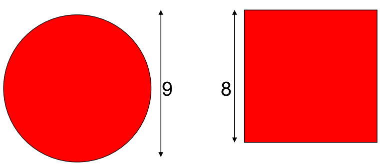
Ancient example of a geometrical algorithm

Rhind papyrus (approx. 1650 BC), copy of an older papyrus of (approx. 1900 BC)

Problem 50: A circular field has diameter 9 khet. What is its area?

Solution: Subtract 1/9 of the diameter which leaves 8 khet. The area is 8 multiplied by 8 or 64 setat.

The Rhind Papyrus



$A = ((8/9)2r)^2$ - approaches π up to 2%

$= 256/81\ r^2$ - "experimental quadrature of the circle "

$= \text{ca } 3.16\ r^2$

Ancient example of Axiomatic Geometry

Some axioms from the "The Elements" of Euclid

Born: about 325 BC

Died: about 265 BC in Alexandria, Egypt

Ancient example of Axiomatic Geometry

Fundamental notions: Point, straight line, plane, incidence relation (" lies on ", " goes through ")

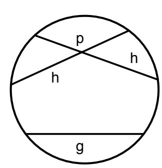
A1: For any two points P and Q there is exactly one straight line g on which P and Q lie.

A2: For each straight line g there is one point, which is not on g.

A3: For each straight line to g and each point P, which is not on g, there is exactly one straight line h, on which P lies and which does not have a common point with g.

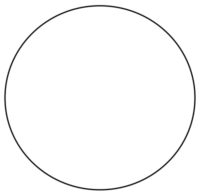
Klein´s model

Question: Is A3 independent of A1 and A2?



Klein´s model

Independence of the parallel axiom



Computational Geometry today

- Back to the historical roots
- Search for simple, robust, efficient algorithms
- Fragmentation into:
  Rather theoretical investigations
  Development of practically useful tools
- Hundreds of papers per year
- Application of algorithmic techniques and data structures
- Efficient solution of fundamental, " simple" problems
- Development of new techniques and data structures
  - Randomization and incremental construction
  - Competitive algorithms

Lecture 1: Introduction

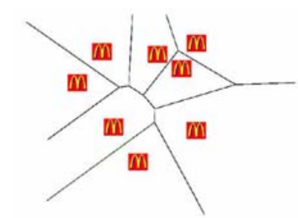
- History: Proof-based, algorithmic, axiomatic geometry, computational geometry today
- Problem fields
- An example: Computing the convex hull:
  1. the “naive approach”
  2. Graham‘s Scan
  3. Lower bound
- Design, analysis, and implementation of geometrical algorithms

Problem fields

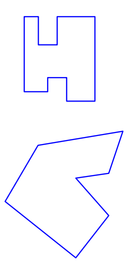
- Typical questions
- Geometrical objects: points, lines, surfaces
- Techniques
- Applications

Finding the nearest fast-food restaurant

Partitioning the plane into areas of equal nearest neighbors

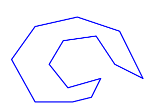


Art gallery problem



How many stationary guards are needed to guard the room?

Watchmen routes

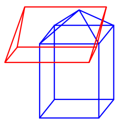


Compute the optimal watchman route for a mobile guard

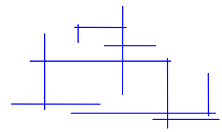
Visibility problems

Hidden-line-elimination

Visible surface computation

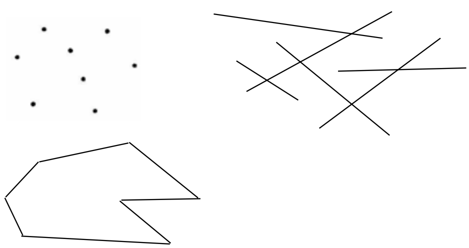


Intersection problems



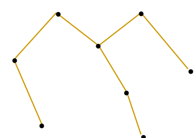
Given a set of line segments, rectangles, polygons, ...: Compute all pairs of intersecting Objects.

Geometric objects: Points, lines, …



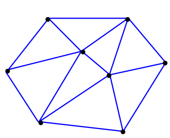
Different algorithms for points

Minimum spanning tree



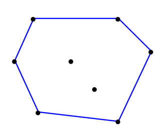
Different algorithms for points

Delauney triangulation

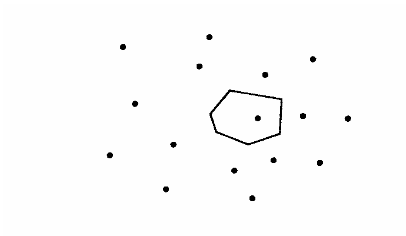


Different algorithms for points

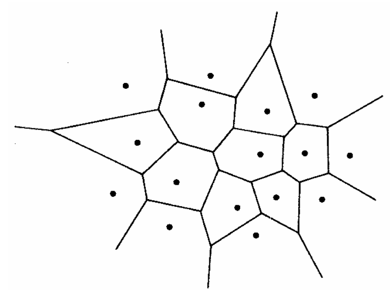
Convex hull



Voronoi Region



Voronoi Diagram



Geometric search

Closest pair

Is it possible to close the gap between $\Omega(n \log n)$ and $O(n^2)$?

Asymptotic bounds are relevant!

Difference between n, n log n and n²

n	n log n	n²
$2^{10} \cong 10^3$	$10 \cdot 2^{10} \cong 10^4$	$2^{20} \cong 10^6$
$2^{20} \cong 10^6$	$20 \cdot 2^{20} \cong 2 \cdot 10^7$	$2^{40} \cong 10^{12}$

Interactive Processing	n log n algorithms	n² algorithms
n = 1000	yes	?
n = 1000000	?	no

Computational geometry has developed new types of algorithms which may solve basic geometric problems efficiently.

Application domains

Computer graphics: 2- and 3-dimensional

Robotics, CAD, CAM

VLSI design

Database systems, GIS

Molecular modelling, ...

Geographical information systems

UNI-Offspring
sofion

Documentation, analysis, and maintenance of gas, water and sewage pipes and telecommunications lines

Robotics

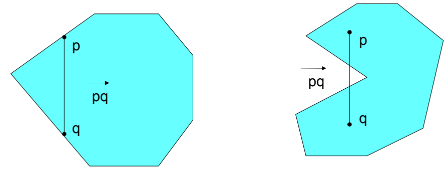
Laserscan robot

Localisation and path-finding in unknown environments.
Example of an On-line scenario of geometrical algorithms

Lecture 1: Introduction

- History: Proof-based, algorithmic, axiomatic geometry, computational geometry today
- Problem fields
- An example: Computing the convex hull:
  1. the “naive approach”
  2. Graham‘s Scan
  3. Lower bound
- Design, analysis, and implementation of geometrical algorithms

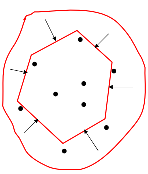
Convex Hulls



Subset of S of the plane is convex, if for all pairs p,q in S the line segment pq is completely contained in S.

The Convex Hull CH(S) is the smallest convex set, which contains S.

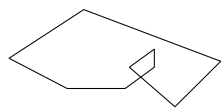
Convex hull of a set of points in the plane



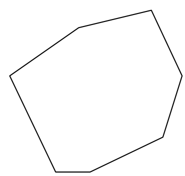
Rubber band experiment

The convex hull of a set P of points is the unique convex polygon whose vertices are points of P and which contains all points from P.

Polygons



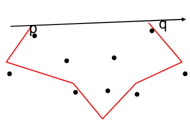
A polygon P is a closed, connected sequence of line segments.



A polygon is simple, if it does not intersect itself.

A simple polygon is convex, if the enclosed area is convex.

Computing the convex hull



Right rule: The line segment pq is part of the CH(P) iff all points of P-{p,q} lie to the right of the line through p and q

Naive procedure

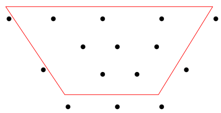
Input : A set P of points in the plane

Output : Convex Hull CH(P)

1. E=∅
2. for all (p, q) from PxP with p ≠ q
3. valid = true
4. for all r in P with r ≠ p and r ≠ q
5. if r lies to the left of the directed line from p to q
6. valid = false
7. if valid then for E = E ∪ { $\overrightarrow{pq}$ }

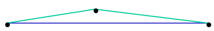
Construct CH(P) as a list of nodes from E
Run time: $O(n)^3$

Degenerate cases



1. Linear dependency: more points are on a line .

   Solution: extended definition by cases

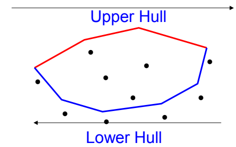


2. Rounding errors due to computer arithmetic (floats).

   Solution: symbolic computation, interval arithmetic

An incremental algorithm

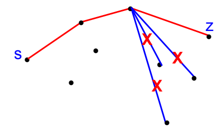
Partitioning of the problem:



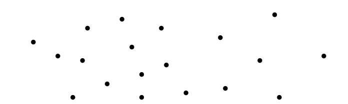
Incremental approach:

Given: Upper hull for $p_1,\ldots,p_{i-1}$

Compute: Upper hull for $p_1,\ldots,p_i$



Computation of UH (Graham Scan)



Fast computation of the convex hull

Input/output: see " naive procedure "

Sort P according to x-Coordinates
LU= $\{p_1, p_2\}$
  for i = 3 to n
    LU = LU $\cup$ { p_i }
    while LU contains more than 2 points and the
      last 3 points in LU do not make a right turn
    do delete the middle of last 3 points
LL= $\{p_n, p_{n-1}\}$
  for i = n-2 to 1
    LL = LL $\cup$ { p_i }
    while LL contains more than 2 points and the
      last 3 points in LL do not make a right turn
    do delete the middle of last 3 points
      delete first and last point in LL
CH(P) = LU $\cup$ LL

Runtime

Theorem: The fast algorithm for computing the convex hull (Graham Scan) can be carried out in time O(n log n).

Proof: (for UH only)
Sorting n points in lexicographic order takes time O(n log n).
Execution of the for-loop takes time O(n).
Total number of deletions carried out in all executions of the while loop takes time O(n).

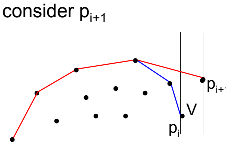
→ Total runtime for computing UH is O(n log n)

Correctness

Proof: for Upper Hull by induction:

(1) $\{p_1, p_2\}$ is a correct UH for a set of 2 points.

(2) Assume: $\{p_1,\ldots,p_i\}$ is a correct UH for i points and consider p_{i+1}

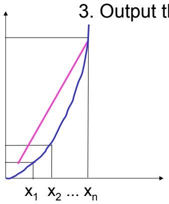


By induction a too high point may lie only in the slab V. This, however, contradicts the lexicographical order of points!

Lower bound

Reduction of the sorting problem to the computation of the convex hull.

1. $x_1, \ldots, x_n \rightarrow (x_1, x_1^2), \ldots, (x_n, x_n^2)$ O(n)
2. Costruct the convex hull for these points
3. Output the points in (counter-)clockwise order



Design, Analysis & Implementation

1. Design the algorithm and ignore all special cases.
2. Handle all special cases and degeneracies.
3. Implementation:
   Computing geometrical objects: best possible
   Decisions (e.g. comparison operations):
   suppose exact (correct) results

Support:

Libraries: LEDA, CGAL

Visualizations: VEGA

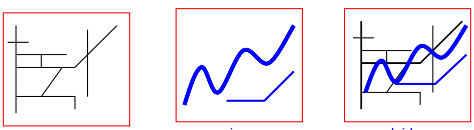
Line Segment Intersection

- Motivation: Computing the overlay of several maps
- The Sweep-Line-Paradigm: A visibility problem
- Line Segment Intersection
- The Doubly Connected Edge List
- Computing boolean operations on polygons

Maps

Motivation

Thematic map overlay in Geographical Information Systems

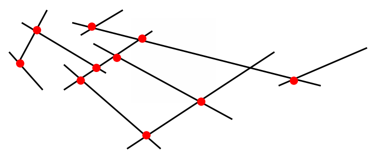


road river overlaid maps

1. Thematic overlays provide important information.
2. Roads and rivers can both be regarded as networks of line segments.

Problem definition

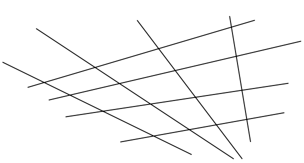
Input: Set $S = \{s_1 \ldots, s_n\}$ of n closed line segments $s_i=\{(x_i, y_i), (x'_i, y'_i)\}$



Output: All intersection points among the segments in S

The intersection of two lines can be computed in time $O(1)$.

Naive algorithm

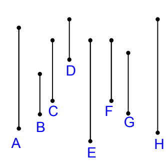


Goal: Output sensitive algorithm!

The Sweep-Line-Paradigm: A visibility problem

Input: Set of n vertcal line segments
Output: All pairs of mutually visible segments

Naive method:

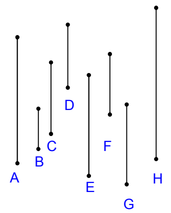


Observation: Two line segments s and s´ are mutually visible iff there is a y such that s and s´are immediate neighbors at y.

Sweep line algorithm

Q is set of the start and end points of the segments in decreasing y-order

T is set of the active line segments



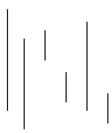
Q	T
.H	H
.D	D!H
.A	A!DH
.F	A!D!F!H
.C	A!C!DFH
.E	ACD!E!FH
D.	AC!EFH
.G	ACEF!G!H
.B	A!B!CEFGH
F.	ABCE!GH
C.	AB!EGH
::	: : :

Algorithm

Initialise Q as set of start and endpoints of segments in decreasing y-order;
Initialise the set of active segments T = ∅;

while Q ≠ ∅ do
  p = Q.Min; remove p from Q;
  if (p start point of segment s)
    T = T ∪ {s}; determine neighbors s´and s´´ of s;
    report (s, s´) and (s,s´´) as visible pairs
  else /* p is end point of segment s */
    determine neighbors s´and s´´ of s;
    report (s´, s´´) as visible pair;
    T = T - {s}

Sweep Line principle



Imaginary line moves in y direction.
Each point is an event.

Input: A set of (iso-oriented objects)
Output: Problem-dependent

Q: object and problem-dependant queue of event points
T: ordered set of the active objects /* status structure */

while Q ≠ ∅ do
  select next event point from Q and remove it from Q;
  update(T);
  report problem-dependent result

Data structures: event queue

Operations to be supported:

Initialisation, min, deletion of points,

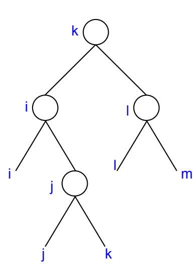
Possible implementation: Balanced search tree of points with order

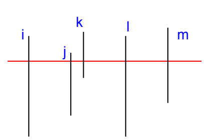
$p < q \Leftrightarrow p_y < q_y$ or ($p_y = q_y$ and $p_x < q_x$)

Initialisation takes time O(m log m) for m items.
Deletion takes time O(log m) with m items in queue.

In most cases a priority queue supporting insertion and min-removal (eg. heap, O(m), O(log m), for initialisation and min-removal) is enough .

Data structures: status structure





Operations to be supported:
Insertion, deletion, searching for neighbors

Possible implementation:

Balanced search tree, O(log n) time

Node values (keys) are used for routing

Runtime analysis

Initialise Q as set of start and endpoints of segments in decreasing y-order;
Initialise the set of active segments T = ∅;
while Q ≠ ∅ do
  p = Q.Min; remove p from Q;
  if (p start point of segment s)
    T = T ∪ {s}; determine neighbors s´and s´´ of s;
    report (s, s´) and (s,s´´) as visible pairs
  else /* p is end point of segment s */
    determine neighbors s´and s´´ of s;
    report (s´, s´´) as visible pair;
    T = T - {s}

Summary

Theorem: For a given set of n vertical line segments all k pairs of mutually visible segments can be reported in time O(n log n).

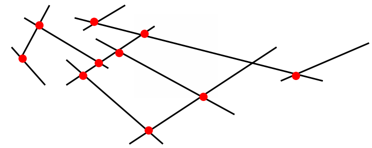
Note: k is O(n)

Line Segment Intersection

- Motivation: Computing the overlay of several maps
- The Sweep-Line-Paradigm: A visibility problem
- Line Segment Intersection
- The Doubly Connected Edge List
- Computing boolean operations on polygons

Line segment intersection

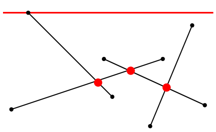
Input: Set $S = \{s_1,...,s_n\}$ of n closed line segments $s_i=\{(x_i, y_i), (x'_i, y'_i)\}$



Output: All intersection points among the segments in S

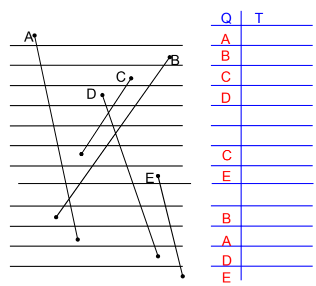
The intersection of two lines can be computed in time O(1).

Sweep line principle



Event queue: upper, lower, intersection points
Status structure: Ordered set of active line segments

Example: Segment Intersection



Data structures: Event Queue Q

Operations: Initialisation (sequence of upper and lower endpoints of segments in decreasing y-order), min-delete, insertion (of intersection points)

Implementation: Balanced search tree with order

$p < q \Leftrightarrow p_y < q_y$ or ($p_y = q_y$ and $p_x < q_x$)

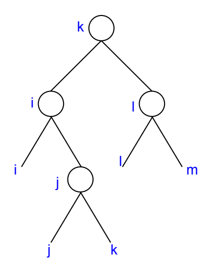
Space: O(n + k), k = #intersections

Time: Initialisation: O(n log n)

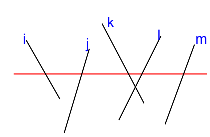
Min-delete: O(log n)

Insertion: O(log n)

Data structures: Status structure T



Balanced search tree



Operations: insertion, deletion, neighbor search, (changing order)

Space: O(n)
Time: O(log n)

Number of operations, total time

n = #segments

k = #intersections

Number of operations on event queueQ: <= 2n+k,

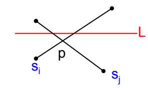
Number of operations on status structureT: <= 2n+k

Result: Total time required to carry out the sweep-line algorithm for computing all k intersections in a set of n line segments is O((n+k) log n).

The sweep-line algorithm is output sensitive!

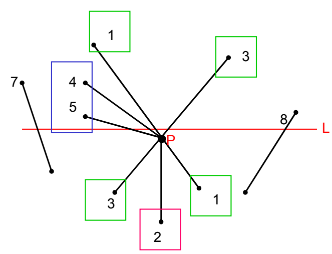
A simple neighborhood lemma

Lemma : Let s_i and s_j be two non-horizontal segments intersecting in a single point p and no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.

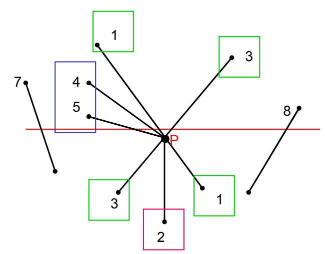


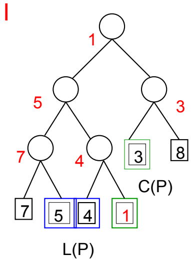
Proof : L is so close to p that s_i and s_j are next to each other. Since s_i and s_j are not yet adjacent at the beginning of the algorithm there is an event q where s_i and s_j become adjacent and tested for intersection.

Handling special cases

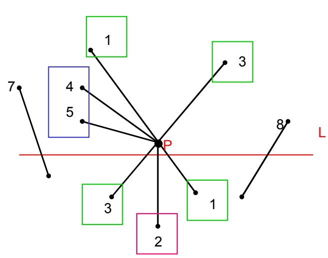


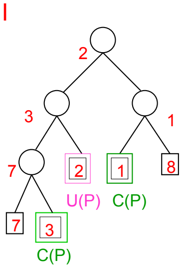
Special cases I





Special cases II





HandleEventPoint(p)

if (L(p) ∪ U(p) ∪ C(p) contains more than 1 segment)
 then {report p as intersection;
  delete L(p) ∪ C(p) from T;
  insert U(p) ∪ C(p) into T;}
if (U(p) ∪ C(p) = {})
 then {Let s_l and s_r be left and right neighbours of p in T
  FindNewEvent(s_l,s_r,p)
 else
  s´ = leftmost segment of U(p) ∪ C(p) in T
  s_l = left neighbour of s´ in T
  FindNewEvent(s_l,s´,p)
  s´´ = rightmost segment of U(p) ∪ C(p)
  s_r = right neighbour of s´´ in T
  FindNewEvent(s´´,s_r,p)

FindNewEvent(s,s´,p)

If (s and s´ intersect below the sweep line L
or on it and to the right of the current event point p)
and (the intersection of s and s´is not yet present in Q)
then insert the intersection point into Q;

Summary

Theorem: Let S be a set of n line segments in the plane. All intersection points in S, with for each intersection point the segments involved in it, can be reported in O(n log n + k log n) time and O(n) space, where k is the size of the output..

k can be reduced to I, I = #intersections

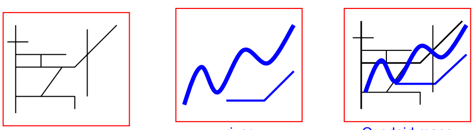
Line Segment Intersection

- Motivation: Computing the overlay of several maps
- The Sweep-Line-Paradigm: A visibility problem
- Line Segment Intersection
- The Doubly Connected Edge List
- Computing boolean operations on polygons

Maps

Motivation

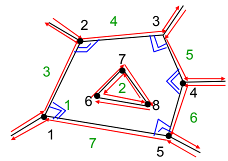
Thematic map overlay in Geographical Information Systems



road river Overlaid maps

1. Thematic overlays provide important information.
2. Roads and rivers can both be regarded as networks of line segments.

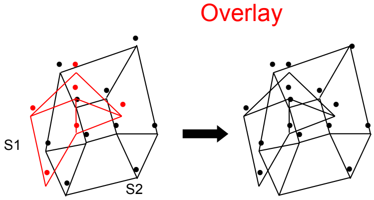
Doubly Connected Edge List



Example
node 1 = { ((1, 2)), 12 }
face 1 = { 15, [67] }
edge 54 = { 5, 45, 1, 43, 15 }

```
3 Records : vertex {
        Coordinates
        IncidentEdge
    };
    face {
        OuterComponent
        InnerComponent
    };
    halfedge {
        Origin
        Twin
        IncidentFace
        Next
        Prev
    };
```

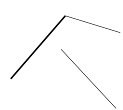

Overlay



U(S1,S2)

Plane Sweep (downward above)
Data structures : Status structure T, Event Queue Q,
Doubly Connected Edge List D
Edges in Q and D are crosswise connected

Overlay



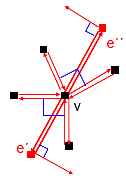
Re-use received from edges, there orientation remains

Updating of T and Q such as segment intersection

2 phases: Edges and corners surfaces

Edges and Corners

Example: Edge of a component cuts nodes of the other



Two halfedge Records e´, e´´ with v as origin generate set twin-pointers and double edges NEXT and PREV at the corner points set and neighbours of e update

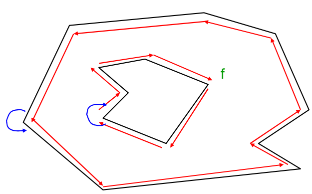
⇒ time O(1 + deg(v)) at a node

⇒ time O(n l og n + k l og n) altogether,

k complexity O(s1, s2)

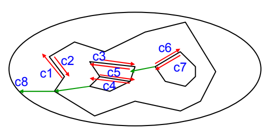
Surfaces

Difference inside and outside by 180°

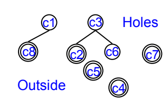


Surfaces

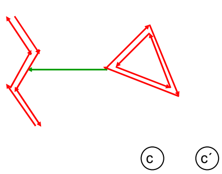
Difference same surface



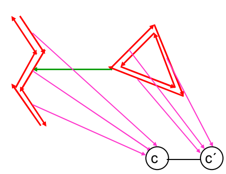
Applies only to linked nodes!



Construction of G

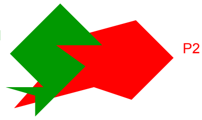


Construction of G



Theorem : Connected components form a surface G can be designed in O(n+k).

Boolean Operations For Polygons



P1 AND P2 (new surfaces in overlap)

P1 OR P2 (all surfaces in overlap)

P1 – P2 (old faces) - (newly generated faces)

Let n = |P1| + |P2|
All 3 operations can be calculated in O(n log n + k log n),
k is output size

LEDA

Library of Efficient Datastructures and Algorithms

http://www.mpi-sb.mpg.de/ LEDA/leda.html

http://www.mpi-sb.mpg.de/ mehlhorn/LEDAbook.html

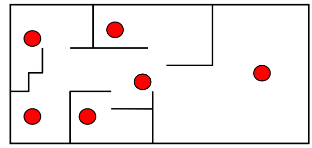
Installed under :

/usr/local/leda/v3.6.1

Polygon Triangulation

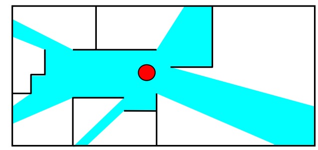
- Motivation: Guarding art galleries
- Art gallery theorem for simple polygons
- Partitioning of polygons into monotone pieces
- Triangulation of y-monotone polygons

Guarding art galleries

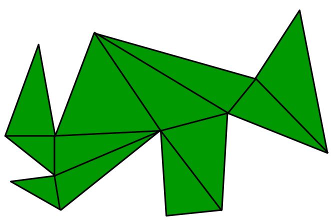


"Art Gallery" Problem

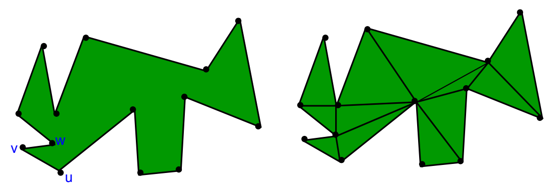
Visibility polygon



Guarding a triangulated polygon



Triangulation of simple polygons



Theorem

Theorem: Every simple polygon admits a triangulation, and any triangulation of a simple polygon with n vertices consists of exactly n-2 triangles.

Proof: By induction on n. Let n>3, and assume theorem is true for all m<n. Let P be polygon with n vertices. We first prove the existence of a diagonal in P. Let v be leftmost vertex of P. Let u and v be two neighboring vertices of v. If $\overline{uw}$ lies in the interior of P we have found a diagonal. Else, there are one or more vertices inside the triangle defined by u, v, and w, or $\overline{\text{the}}$ diagonal uw. Let v´ be the $\overline{\text{far}}$thest vertex from uw. The segment connecting v´ to v cannot intersect an edge of p (contradicts the defin$\overline{\text{iti}}$on of v´). Hence vv´ is a diagonal.

Continuation of proof

So a diagonal exists. Any diagonal cuts P in two simple sub-polygons P_1 and P_2. Let m_1 be the number of vertices of P_1 and m_2 the number of vertices of P_2. Both m_1 and m_2 must be smaller than n, so by induction P_1 and P_2 can be triangulated so P can be triangulated as well.

Now we have to prove any triangulation of P contains n-2 triangles. Consider an arbitrary diagonal in some triangulation T_p. This diagonal cuts P into 2 subpolygons with m_1 and m_2 vertices. Every vertex of P occurs in exactly one of 2 subpolygons. Hence $m_1+m_2 = n+2$. So by induction any triangulation of P_i contains m_i-2 triangles $\Rightarrow (m_1-2) + (m_2-2) = n-2$ triangles.

Number of triangles in any triangulation of a simple polygon with n vertices.

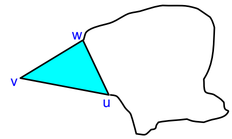
Case 1: n=3

Case 2: n>3

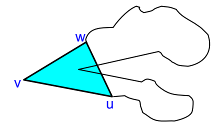
Proof of the existence of diagonals in P

Consider leftmost vertex v of P

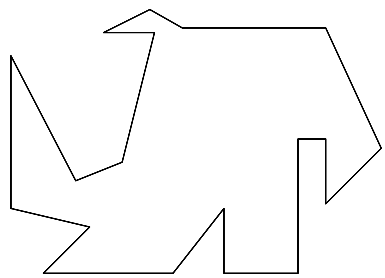
Case 1: uw completely in P



Case 2: uw not completely in P



Proof of the existence of a diagonal in P

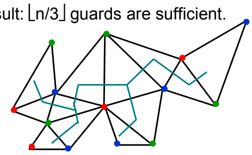


Upper and lower bounds for the number of guards

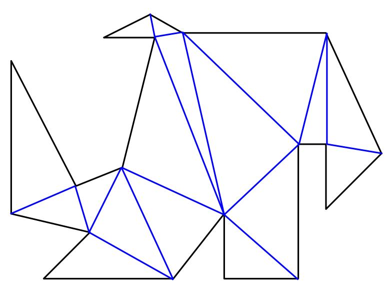
We know that for any simple polygon with n vertices (n-2)guards are always enough.

But can we do better?

Idea: Compute a 3-coloring of the vertices and place guards on a color.
Result: $\lfloor n/3 \rfloor$ guards are sufficient.



Example



Theorem

Theorem: Each simple polygon is 3-colorable.

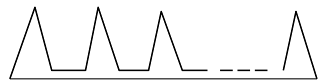
Proof: Dual graph is a binary tree, this means that we can find a 3-coloring using a simple DFS.

Corollary: ⌊n/3⌋ guards are always sufficient to guard a simple Polygon with n vertices.

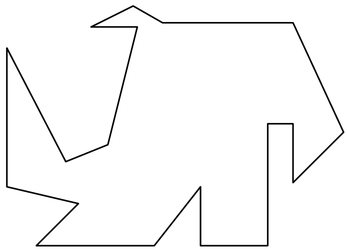
Art gallery theorem

Theorem: For a simple polygon with n vertices, ⌊n/3⌋ cameras are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the cameras.

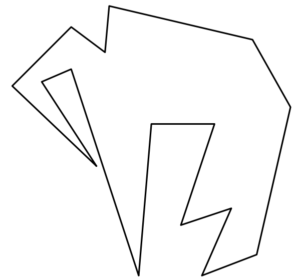
Proof: Worst-case example.



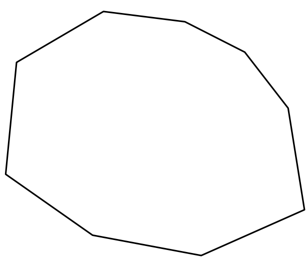
Triangulation (Naive)



Triangulation (Naive)



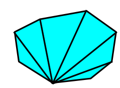
Triangulation of a convex polygon



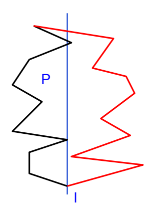
l-Monotone

Convex polygons are easy to triangulate.

Unfortunately the partition into convex pieces is just as difficult as the triangulation.



l-monotone



A simple polygon is called monotone w.r.t. a line l if for any line l´ perpendicular to l the intersection of the polygon with l´ is connected (y-monotone, if l = y-axis).

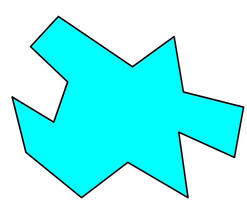
Observation: if P is y-monotone then P Consists of two y-monotone chains.

Two steps for triangulation

1. Divide P into y-monotone parts $P_1,\ldots,P_k$

2. Triangulate $P_1,\ldots,P_k$

Split and merge vertices



= start vertex

= end vertex

= regular vertex

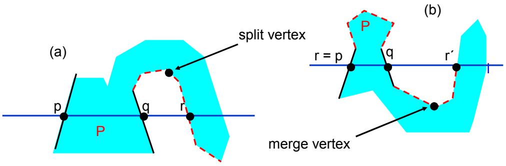
= split vertex

= merge vertex

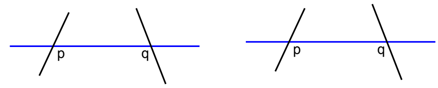
Lemma: A polygon is y-monotone if it has no split vertices or merge vertices.

Proof: Suppose P is not y-monotone ⇒ there is a horizontal line l that intersects P in more than one connected component.

We show that P must have at least one split or merge vertex:



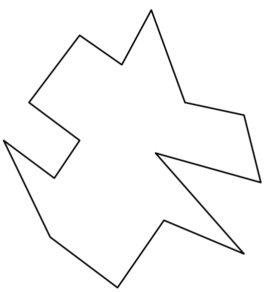
Start with q to r achieved (go upward)



(a) r ≠ p ⇒ there exists a split node between q and r.

(b) r = p ⇒ there exists a r´ (go downward) and thus a merge node.

Five types of vertices



= start vertex

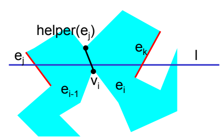
= end vertex

= regular vertex

= split vertex

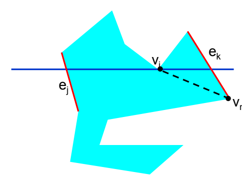
= merge vertex

Removal of Split and Merge nodes

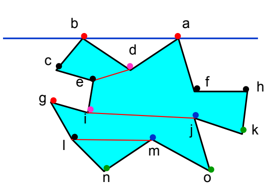


helper(e_j) is lowest vertex above the sweep line such that the horizontal segment connecting the vertex to e_j lies inside P.

Merge-nodes are split nodes in reverse. v_i is the new helper of e_j. We would like to connect v_i to the highest vertex below the sweep line in between e_j and e_k



Example



v		T	helper
a		ad	ad = a
b		ad,bc	bc = b
c		ad,ce	ce = c
d		ce	ce = d
e	!	ei	ei = e
f		ei	
g		ei,gl	gl = g
h		ei,gl	
i		gl	gl = i
j	!	gl,jo	jo = gl = j
k		gl	
l		ln	ln = l
m	!	ln,mo	ln = mo = m
n		mo	
o			

Algorithm: MakeMonotone

Input: A simple polygon P stored in a doubly-connected edge list D

Output: A partitioning of P into monotone sub-polygons, stored in D

Construct a priority queue Q on the vertices of P.
Initialize an empty binary search tree T.
while Q is not empty
    do remove the vertex v_i with highest priority from Q
call appropriate procedure to handle the vertex.

Handling start, end and split vertices

HandleStartVertex(v_i): $T = T \cup \{e_i\}$, helper(e_i) = v_i

HandleEndVertex(v_i): if (helper(e_{i-i}) is merge vertex)
then insert diagonal connecting v_i to helper(e_{i-1}) in D.
$T = T - \{e_{i-1}\}$

HandleSplitVertex(v_i): Search in T to find the edge directly left of v_i

Insert the diagonal connecting v_i to helper(e_j) in D.

helper(e_j) = v_i

Insert e_i in T and set helper(e_i) to v_i

Handling merge vertices

HandleMergeVertex(v_i) : if helper(e_{i-1}) is a merge vertex
then Insert diagonal connecting v_i to helper(e_{i-1}) in D.

Delete e_{i-1} from T.

Search in T to find the edge e_j left of v_i .

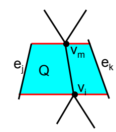
if helper(e_j) is a merge vertex
then Insert diagonal connecting v_i to helper(e_j) in D.

helper(e_j) = v_i

Handling regular vertices

HandleRegularVertex(v_i) : if the interior of P lies to the right of v_i
then if helper(e_{i-1}) is a merge vertex
then Insert the diagonal connecting v_i to helper(e_{i-1}) in D
delete e_{i-1} from T.
insert e_i in T and set helper(e_i) to v_i.
else search in T to find the edge e_j left of v_i
if helper(e_j) is a merge vertex
then insert the diagonal connecting v_i to helper(e_j) in D
helper(e_j) = v_i

Correctness of HandleSplitVertex



Consider a segment $\overline{v_m v_i}$ that is added when v_i is reached by HandleSplitVertex. Let e_j be the edge to left of v_i, and let e_k be the edge to right of v_i . helper(e_j) = v_m when we reach v_i .
Argument : $\overline{v_m v_i}$ does not intersect an edge of P
Consider the quadrilateral Q, there are no vertices of P inside Q, else v_m would not be helper of e_j .
Suppose an edge of P intersects $\overline{v_m v_i}$ then it would have to intersect a segment connecting v_i to e_j but this is impossible.
Since there are no vertices of P inside Q, no edge of P can intersect $\overline{v_m v_i}$

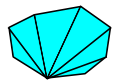
Theorem

A simple polygon with n vertices can be partitioned into y-monotone polygons in O(n log n) time with an algorithm that uses O(n) storage.

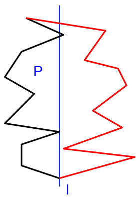
l-Monotone

Convex polygons are easy to triangulate.

Unfortunately the partition into convex parts is just as difficult as the triangulation.



l-monotone



A simple polygon is called monotone w.r.t. a line l if for any line l´ perpendicular to l the intersection of the polygon with l´ is connected (y-monotone, if l = y-Axis).

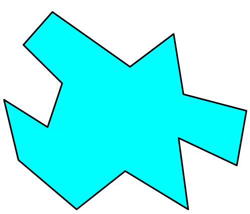
Observation: P is y-monotone.

Two steps for triangulation

1. Divide P into y-monotone parts $P_1,\ldots,P_k$

2. Triangulate $P_1,\ldots,P_k$

Split and Merge Vertices



= start vertex

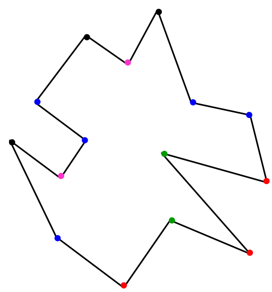
= end vertex

= regular vertex

= split vertex

= merge vertex

Five Types of Vertices



● = start vertex

● = end vertex

● = regular vertex

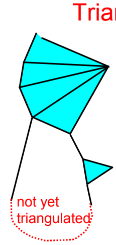
● = split vertex

● = merge vertex

Theorem

A simple polygon with n vertices can be partitioned into y-monotone polygons in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.

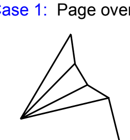
Triangulation of y-monotone Polygon



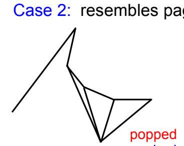
Idea: Fan so long build to convexity hurts alternation from right and left side

Implementation: Scan-line uses stack as data structure

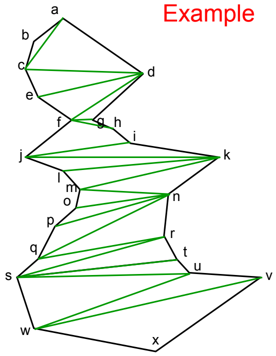
Case 1: Page overflows



Case 2: resembles page



Example

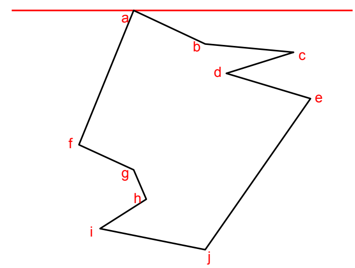


Batches : ba

c : ba → ca

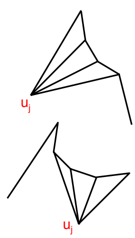
d : ca → dc

e : dc → ed



Implementation

```
S.push(u1), S.push(u2)
for j = 3,...,n-1
  if (side(uj) ≠ side(S.top))
    while (S ≠ ∅) v = S.pop, diag(uj,v)
      S.push(uj-1)
      S.push(uj)
  else
    while (diag(S.top, uj) in P)
      diag(S.top, uj)
      S.pop
      S.push(last)
      S.push(uj)
```



Theorem: time $O(n)$

Proof: number of pops < number of pushes

Theorem

Theorem: A strictly y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Theorem: A simple polygon with n vertices can be triangulated in $O(n \log n)$ with an algorithm that uses $O(n)$ storage.

Theorem: A planar subdivision with n vertices in total can be triangulated in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.

Computational Geometry Algorithms Library

CGAL

http://www.cs.uu.nl/CGAL

Kernel

2D/3D point, vector, direction, segment, ray, line, dD point, triangle, bounding box,iso-rectangle, circle, plane, tetrahedron, predicates, affine transformations, intersection and distance calculation

Basic Library

half edge data structure, topological map, planar map, polyhedron, Boolean operations on polygons, planar map overlay, triangulation, Delauney triangulation, 2D/3D convex hull, and 2D extreme points, smallest enclosing circle/sphere and ellipse, maximum inscribed k-gon, and other optimizations, range tree, segment tree, kD tree

Linear Programming

Overview

- Formulation of the problem and example
- Incremental, deterministic algorithm
- Randomized algorithm
- Unbounded linear programs
- Linear programming in higher dimensions

Problem description

Maximize $c_1x_1 + c_2x_2 + \dots + c_dx_d$

Subject to the conditions:

$$\begin{array}{ccc} a_{1,1}x_1 + \dots a_{1,d}x_d & \leq & b_1 \\ a_{2,1}x_1 + \dots a_{2,d}x_d & \leq & b_2 \\ \vdots & \vdots & \vdots \\ a_{n,1}x_1 + \dots a_{n,d}x_d & \leq & b_n \end{array}$$

Linear program of dimension d:

$$\vec{c} = (c_1, c_2, \dots, c_d)$$
$$h_i = \{(x_1, \dots, x_d)\ ;\ a_{i,1}x_1 + \dots + a_{i,d}x_d \leq b_i\}$$

l_i = hyperplane that bounds h_i (straight lines, if d=2)

$$H = \{h_1, \dots, h_n\}$$

Example

Production of two goods A and B using four raw materials

Value of A: 6 CU, value of B: 3 CU

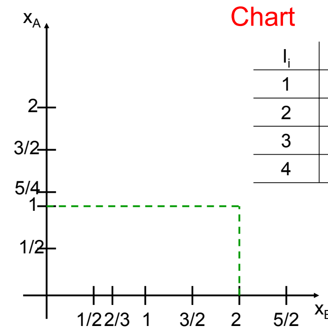
	Rm1	Rm2	Rm3	Rm4
Prod A	2	2	6	2
Prod B	4	1	2	2
Reserve	5	2	4	3

Maximize profit: $f_c(x) = 6x_A + 3x_B$ under the conditions:

$2x_A + 4x_B \leq 5$
$2x_A + 1x_B \leq 2$
$6x_A + 2x_B \leq 4$
$2x_A + 2x_B \leq 3$
$x_A, x_B \geq 0$

	$x_A = 0, x_B$	$x_A, x_B = 0$

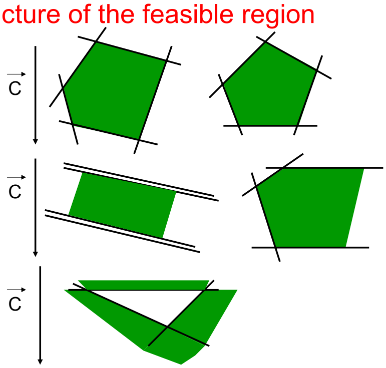
Chart



l_i	(x_A, c)	(x_A, x_B)
1	(0, 5/4)	(5/2, 0)
2	(0, 2)	(1, 0)
3	(0, 2)	(2/3, 0)
4	(0, 3/2)	(3/2, 0)

Structure of the feasible region

1. Bounded
2. Unbounded
3. Empty



Result

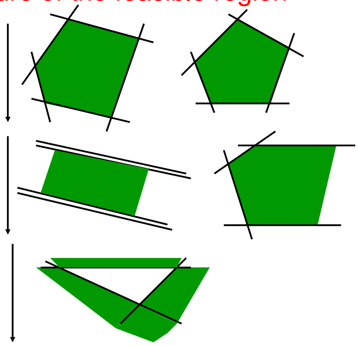
Four possibilities for the solution of a linear program

1. A vertex of the feasible region is the only solution.
2. One edge of the feasible region contains all solutions.
3. There are no solutions.
4. The feasible region is unbounded toward the direction of optimization.

In case 2: Choose the lexicographically minimum solution = > corner

Structure of the feasible region

1. Bounded $\vec{C}$
2. Unbounded $\vec{C}$
3. Empty $\vec{C}$



Linear Programming

Overview

- Formulation of the problem and example
- Incremental, deterministic algorithm
- Randomized algorithm
- Unbounded linear programs
- Linear programming in higher dimensions

Problem description

Maximize $c_1x_1 + c_2x_2 + \dots + c_dx_d$

Subject to the conditions:

$$a_{1,1}x_1 + \dots a_{1,d}x_d \leq b_1$$
$$a_{2,1}x_1 + \dots a_{2,d}x_d \leq b_2$$
$$\vdots \quad \vdots \quad \vdots$$
$$a_{n,1}x_1 + \dots a_{n,d}x_d \leq b_n$$

Linear program of dimension d:

$$\vec{c} = (c_1,c_2,\dots,c_d)$$
$$h_i = \{(x_1,\dots,x_d) \,;\, a_{i,1}x_1 + \dots + a_{i,d}x_d \leq b_i\}$$

l_i = hyperplane that bounds h_i (straight lines, if d=2)

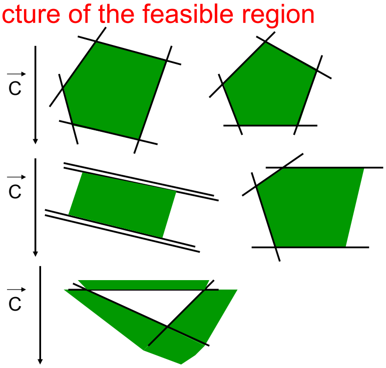
$$H = \{h_1, \dots , h_n\}$$

Structure of the feasible region

1. Bounded

2. Unbounded

3. Empty



Bounded linear programs

Assumption :

Algorithm UnboundedLP(H, $\vec{c}$) yields either

a) a ray in $\cap$ H, which is unbounded towards $\vec{c}$, or

b) two half planes h_1 and h_2, so that $h_1 \cap h_2$ is bounded towards $\vec{c}$, or

c) the answer, that LP(H, $\vec{c}$) has no solution, because the feasible region is empty.

Incremental algorithm

Let $C_2 = h_1 \cap h_2$

Remaining half planes: $h_3,\dots, h_n$

$C_i = C_{i-1} \cap h_i = h_1 \cap \dots \cap h_i$

Compute-optimal-vertex (H, $\vec{c}$)

    $v_2 := l_1 \cap l_2$; $C_2 := h_1 \cap h_2$
    for i := 3 to n do
        $C_i := C_{i-1} \cap h_i$
        v_i := optimal vertex of C_i

$C_2 \supseteq C_3 \supseteq C_4 \dots \supseteq C_n = C$

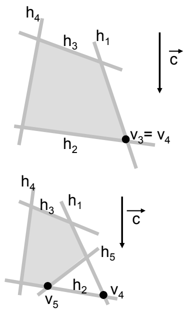
$C_i = \varnothing \Rightarrow C = \varnothing$

Optimal Vertex

Lemma 1: Let $2 < i \leq n$, then we have :

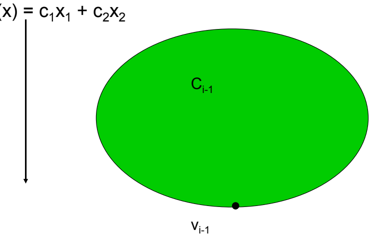
1. If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.
2. If $v_{i-1} \notin h_i$, then either $C_i = \varnothing$ or $v_i \in l_i$, where l_i is the line bounding h_i.

Optimal Vertex



Next optimal vertex

$f_c(x) = c_1x_1 + c_2x_2$



Algorithm 2D-LP

Input: A 2-dimensional Linear Program (H, $\vec{c}$)

Output: Either one optimal vertex or ∅ or a ray along which (H, $\vec{c}$) is unbounded.

if UnboundedLP(H, $\vec{c}$) reports (H, $\vec{c}$) is unbounded or infeasible

then return UnboundedLP(H, $\vec{c}$)

else report $h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$

let $h_3,...,h_n$ be the remaining half-planes of H

for i:= 3 to n do

if $v_{i-1} \in h_i$ then $v_i := v_{i-1}$

else $S_{i-1} := H_{i-1} \cap^* l_i$

v_i := 1-dim-LP(S_{i-1}, $\vec{c}$)

if v_i not exists then return ∅

return v_n

Running time: $O(n^2)$

Algorithm 1D-LP

Find the point x on l_i that maximizes $\vec{cx}$, subject to the constraints $x \in h_j$, for $1 \le j < i-1$

Observation: $l_i \cap h_j$ is a ray

Let $S_{i-1} := \{ h_1 \cap l_i, ..., h_{i-1} \cap l_i\}$

Algorithm 1D-LP{S_{i-1}, $\vec{c}$ }

$p_1 = s_1$

for j := 2 to i-1 do $p_j = p_{j-1} \cap s_j$

if $p_{i-1} \neq \emptyset$ then

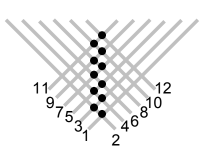
return the optimal vertex of p_{i-1} else

return ∅

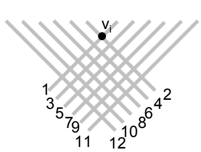
Time: $O(i)$

Addition of halfplanes in different orders

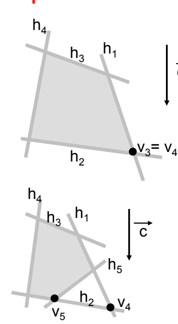
Bad



Good



Optimal vertex



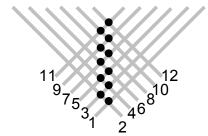
Algorithm 2D-LP

Input: A 2-dimensional Linear Program $(H, \vec{c})$
Output: Either one optimal vertex or ∅ or a ray along which $(H, \vec{c})$ is unbounded.

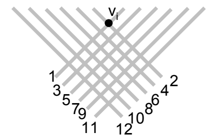
if UnboundedLP$(H, \vec{c})$ reports $(H, \vec{c})$ is unbounded or infeasible
    then return UnboundedLP$(H, \vec{c})$
    else report $h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$
    let $h_3, \ldots, h_n$ be the remaining half-planes of H
    for i:= 3 to n do
        if $v_{i-1} \in h_i$ then $v_i := v_{i-1}$
            else $S_{i-1} := H_{i-1} \cap^* l_i$
            $v_i :=$ 1-dim-LP$(S_{i-1}, \vec{c})$
            if v_i not exists then return ∅
    return v_n
Running time: $O(n^2)$

Addition of halfplanes in different orders

Bad



Good



Linear Programming

Overview

- Formulation of the problem and example
- Incremental, deterministic algorithm
- Randomized algorithm
- Unbounded linear programs
- Linear programming in higher dimensions

Algorithm 2D-LP

Input: A 2-Dimensional Linear Program $(H, \vec{c})$
Output: Either one optimal vertex or ∅ or a ray along which $(H, \vec{c})$ is unbounded.

if UnboundedLP$(H, \vec{c})$ reports $(H, \vec{c})$ is infeasible
  then return UnboundedLP$(H, \vec{c})$
  else $h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$
    $h_3, \ldots, h_n$:= remaining half-planes of H
for i:= 3 to n do
  if $v_{i-1} \in h_i$
   then $v_i := v_{i-1}$
   else $S_{i-1} := H_{i-1} \cap^* l_i$
    $v_i :=$ 1-dim-LP$(S_{i-1}, \vec{c})$
  if v_i not exists then return ∅
return v_n

Running time: $O(n^2)$

New problem

Find the point x on l_i that maximizes $c\vec{x}$, subject to the constraints $x \in h_j$, for $1 \le j < i-1$

Observation: $l_i \cap h_j$ is a ray

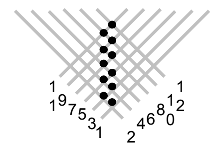
Let $S_{i-1} := \{ h_1 \cap l_i, \ldots, h_{i-1} \cap l_i \}$

1. 1-dim-LP$\{S_{i-1}, \vec{c}\}$
2. $p_1 = s_1$
3. for j := 2 to i-1 do
4. $p_j = p_{j-1} \cap s_j$
5. if $p_{i-1} \neq \emptyset$ then
6. return the optimal vertex of p_{i-1} else
7. return ∅

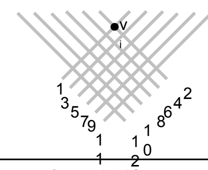
Time: $O(i)$

Sequences

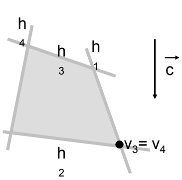
Bad

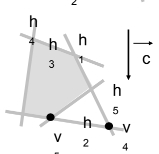


Good



Optimal Vertex





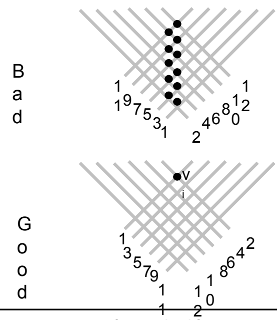
Algorithm 2D-LP

Input: A 2-Dimensional Linear Program $(H, \vec{C})$
Output: Either one optimal vertex or ∅ or a ray along which $(H, \vec{C})$ is unbounded.

if UnboundedLP$(H, \vec{C}) \neq \{h, h'\}$
then return UnboundedLP$(H, \vec{C})$
  $h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$
  $h_3, \ldots, h_n$:= remaining half-planes of H
  for i:= 3 to n do
   if $v_{i-1} \in h_i$
    then $v_i := v_{i-1}$
    else $S_{i-1} := H_{i-1} \cap^* l_i$
     $v_i :=$ 1-dim-LP$(S_{i-1}, \vec{C})$
   if v_i does not exist then return ∅
return v_n

Running time: $O(n^2)$

Sequences



Algorithm 2D-LP

Input: A 2-Dimensional Linear Program $(H, \vec{C})$
Output: Either one optimal vertex or ∅ or a ray along which $(H, \vec{C})$ is unbounded.

if UnboundedLP$(H, \vec{C}) \neq \{h, h'\}$ then
return UnboundedLP$(H, \vec{C})$
$h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$
$h_3,\ldots,h_n$:= remaining half-planes of H
compute a random permutation $h_3, \ldots, h_n$
  for i:= 3 to n do
    if $v_{i-1} \in h_i$ then $v_i := v_{i-1}$
    else $S_{i-1} := H_{i-1} \cap^* l_i$
      v_i := 1-dim-LP$(S_{i-1}, \vec{C})$
    if v_i does not exist then
      return ∅
return v_n

Running time: $O(n^2)$

Randomization

Theorem: The 2-dimensional linear programming problem with n constraints can be solved in O(n) randomized expected time using worst-case linear storage.

Random Variable x_i

$$X_i = \begin{cases} 1 & \text{if } v_{i-1} \text{ in } h_i \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_{i=3}^{n} O(i) \bullet X_i$$

$$E\left[\sum_{i=3}^{n} O(i) \bullet X_i\right] = \sum_{i=3}^{n} O(i) \bullet E[x_i]$$

$E[x_i]$ is the probability that $v_{i-1} \notin h_i$

Algorithm 2D-LP

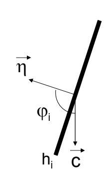
Input: A 2-Dimensional Linear Program $(H, \vec{C})$
Output: Either one optimal vertex or ∅ or a ray along which $(H, \vec{C})$ is unbounded.

if UnboundedLP$(H, \vec{C}) \neq \{h, h'\}$
 then return UnboundedLP$(H, \vec{C})$
  $h_1 := h$; $h_2 := h'$; $v_2 := l_1 \cap l_2$
  $h_3,\ldots,h_n$:= remaining half-planes of H
  for i:= 3 to n do
    if $v_{i-1} \in h_i$
     then $v_i := v_{i-1}$
     else $S_{i-1} := H_{i-1} \cap^* l_i$
      v_i:= 1-dim-LP$(S_{i-1}, \vec{C})$
    if v_i does not exist then return ∅
 return v_n

Running time: $O(n^2)$

Unbounded Linear Programs

$\vec{\eta}$:=The outward normal of h_i



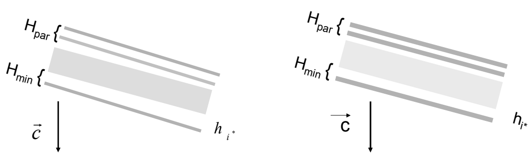
φ_i := The smaller angle that $\vec{\eta}$ makes with $\vec{c}$

I_{min}, an index with

$\varphi_{i\,min} = \min \varphi_j, 1 \leq j \leq n$

$H_{min} := \{ h_j \in H \mid \vec{\eta}_j = \vec{\eta}_{min}\}$

$H_{par} := \{ h_j \in H \mid \vec{\eta}_j = -\vec{\eta}_{min}\}$



Lemma

Let $H = \{h_1, h_2, \ldots, h_n\}$ be a set of half-planes.

Assuming that $\cap(H_{min} \cup H_{par})$ is not empty.

1. If $l_{i^*} \cap h_{j^*}$ is unbounded in the direction $\vec{c}$ for every half-plane h_j in the set $H \setminus (H_{min} \cup H_{par})$, then $(H, \vec{c})$ is unbounded along a ray contained in l_{i^*}.
2. If $l_{i^*} \cap h_{j^*}$ is bounded in the direction $\vec{c}$ for some h_{j^*} in $H \setminus (H_{min} \cup H_{par})$, then the linear program $(\{h_{i^*}, h_{j^*}\}, \vec{c})$ is bounded.

Algorithm UNBOUNDEDLP

Input: A 2-Dimensional Linear Program $(H, \vec{C})$

Output: Either one optimal vertex or $\varnothing$ or a ray along which $(H, \vec{C})$ is unbounded.

1. For each half plane $h_i \in H$ compute $\varnothing_i$
2. Let hi be half plane with $\varnothing_j = \min \varnothing_j, 1 \le j \le n$
3. $H_{min} := \{ h_j \in H \mid \vec{\eta}_j = \vec{\eta}_{min}\}$
4. $H_{par} := \{ h_j \in H \mid \vec{\eta}_j = -\vec{\eta}_{min}\}$
5. $\hat{H} = H \setminus (H_{min} \cup H_{par})$, compute intersection in $H_{min} \cup H_{par}$
6. If the intersection is empty
   then report $(H, \vec{C})$ is feasible
   else Let $h_i \in H_{min}$ be the half-plane whose line bound the intersection
   if there is half plane $h_{j^*} \in \hat{H}$ such that $l_{i^*} \cap h_{j^*}$ bounded in $\vec{C}$
   then report $(\{h_{i^*}, h_{j^*}\}, \vec{C})$ is bounded
   else report is bounded along $l_{i^*} \cap (\cap \hat{H})$

Higher Dimensions

Let $h_1, \ldots, h_d$ H be the d certificate half-spaces that UNBOUNDEDLP returns.

$C_i := h_1 \cap h_2 \cap \ldots \cap h_i$

Lemma: Let $d < i \le n$, and let C_i be defined as above.

1. If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$
2. If $v_{i-1} \notin h_i$, then either $C_i = \varnothing$ or $v_i \in g_i$, where g_i is the hyperplane that bounds h_i.

Algorithm RANDOMIZEDLP

Input : A linear program $(H, \vec{c})$.

Output : Either one optimal vertex or $\varnothing$ or a ray along which $(H, \vec{c})$ is unbounded.

if UNBOUNDEDLP$(H, \vec{c})$ reports $(H, \vec{c})$ is unbounded
  then Report the information and, ray along which $(H, \vec{c})$ is unbounded.
else Let $h_1, \ldots, h_d \in H$ he the certificate halfplanes returned by UNBOUNDEDLP, and let v_d be their vertex of intersection
  Compute a random permutation $h_{d+1}, \ldots, h_n$
  for i = d+1 to n
    do if $v_{i-1} \in h_i$
      then $v_i = v_{i-1}$
      else v_i = the point p on gi that maximizes $f_{\vec{c}}(p)$
        if p does not exist
          then report infeasible and quit.
  Return v_n

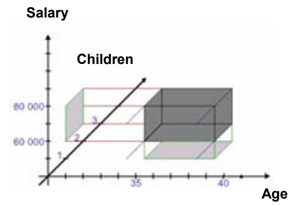
Theorem

The d-dimensional linear programming problem with n constraints can be solved in O(d!n) expected time using linear storage.

Orthogonal Range Searching

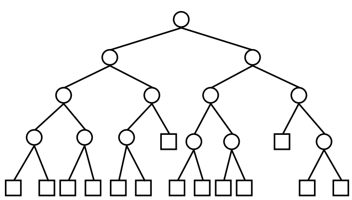
1. Linear Range Search : 1-dim Range Trees
2. 2-dimensional Range Search : kd-trees
3. 2-dimensional Range Search : 2-dim Range Trees
4. Range Search in Higher Dimensions

Range search

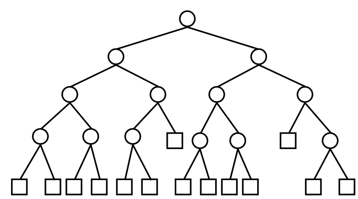


Input: Set of data points in d-space,
orthogonal (iso-oriented) query range R
Output: All points contained in R

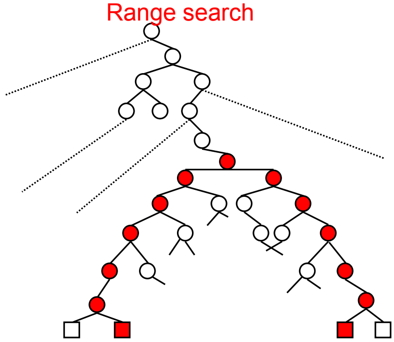
Binary search tree (1-dimensional)



Binary leaf search tree



Range search



1-Dimensional range query

Finding the split node for query range [x, x´]

FindSplitNode (T, x, x´)

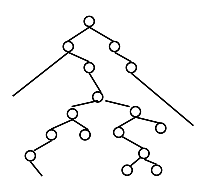
v = root (T)

while not leaf(v) && (x´ ≤ v.x || x > v.x)

  if (x´ ≤ v.x) then v = left(v)

       else v = right(v)

return v



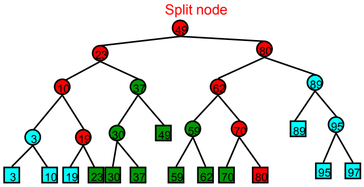
Running time : O(log n)

Note : Only O(log n) subtrees

fall into the query range.

Example – Binary Search Tree

Query range: [22, 77]



Algorithm 1-d-range-search

1DRangeQuery (T, [x, x´])

v_{split} = FindSplitNode (T, x, x´)

if (leaf (v_{split}) && v_{split} in R=[x, x´])

  then write v_{split};

  return;

v = left-child(v_{split});

while (not leaf (v))

    if (x ≤ v.x)

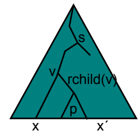
      then write Subtree (right-child(v));

        v= left-child(v);

      else v = right-child(v)

if (v in R) write v ;

v = right-child(v_{split}) ...



Theorem

A 1-dim range query in a set of n points can be answered in time O(log n + k) using a 1-d-range tree, where k is the number of reported points which fall into the given range.

Proof :
FindSplitNode: O(log n)
Leaf search: O(log n)
The number of green nodes is O(k), since number of internal nodes is O(k)
⇒ O((log n)+k) total time.

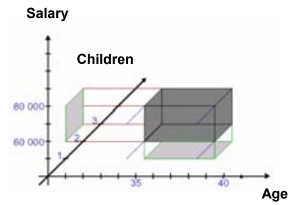
Summary

Let P be a set of n points in 1-dimensional space. The set P can be stored in a balanced binary search tree, which uses O(n) storage and has O(n log n) construction time, such that the points in a query range can be reported in time O(k + log n) , where k is the number of reported points.

Orthogonal Range Searching

1. Linear Range Search : 1-dim Range Trees
2. 2-dimensional Range Search : kd-trees
3. 2-dimensional Range Search : 2-dim Range Trees
4. Range Search in Higher Dimensions

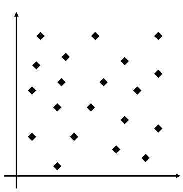
Range search



Input: Set of data points in d-space, orthogonal (iso-oriented) query range R

Output: All point contained in R

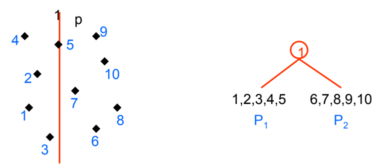
2 – dimensional range search



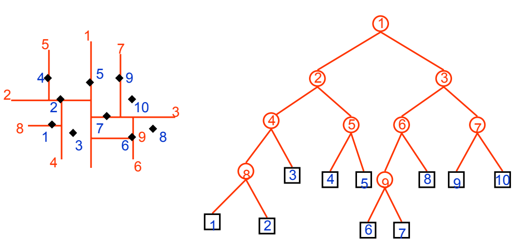
Assumption :

No two points have the same x- or y-coordinates

Construction of kd-trees



Construction of kd-trees



Algorithm for building 2d-trees

BuildTree (P, depth)

if (|P| = 1) return leaf(P)

if (depth even) split P into P_1,P_2 through vertical median

else split P into P_1,P_2 through horizontal median

v_1 = BuildTree (P1, depth + 1)

v_2 = BuildTree (P_2, depth + 1)

return (v_1 , median , v_2)

Analysis

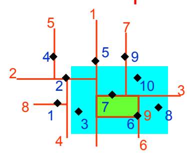
Theorem: The algorithm for constructing a 2d-tree uses O(n) storage and can be carried out in time O(n log n)

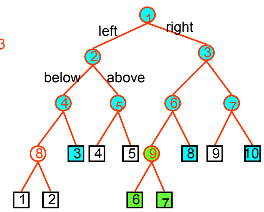
Proof:
Space: 2d-tree is a binary tree with n leaves.
Time:

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

$$\begin{aligned} T(n) &\le cn + 2T(n/2) \\ &\le cn + 2(cn/2 + 2T(n/4)) \\ &\le \ldots \\ &= O(n \log n) \end{aligned}$$

Nodes represent regions

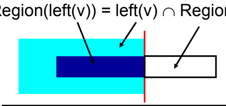




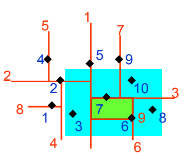
Regions: Region(5) is left of 1 and above 2

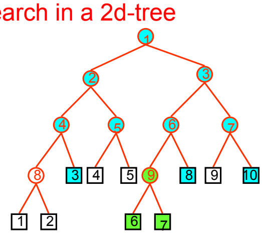
Incremental .... :

Region(left(v)) = left(v) ∩ Region(v)



Algorithm search in a 2d-tree





SearchTree(v,R)

if (leaf(v) && v in R) then write v; return

if (Region(left(v)) in R) then write Subtree(left(v)), return

if (Region(left(v)) ∩ R <> ∅) then SearchTree(left(v), R)

if (Region(right(v)) in R) then write Subtree(right(v)),return

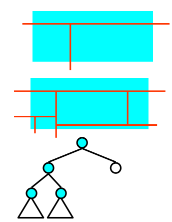
if (Region(right(v)) ∩ R <> ∅) then SearchTree(right(v),R)

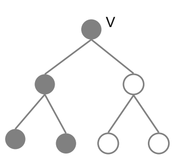
Analysis algorithm search in 2d-tree

Lemma : A query with an axis-parallel rectangle in a 2d-tree storing n points can be performed in $O(\sqrt{n} + k)$ time, where k is the number of reproted points.

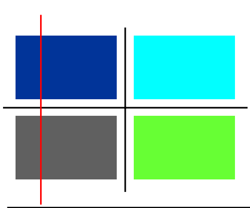
Proof : B = # of blue nodes, G = # of green nodes
G(n) = O(k).
B(n) ≤ # of vertical intersection regions V + # of horizontal intersection regions H
Line l intersects either the region to left of root(T) or to the right.
This gives the following recursion:

$$V(n) = O(1), \quad \text{if } n = 1$$
$$= 2 + 2V(n/4), \text{ if } n > 1$$
$$V(n) = 2 + 4 + 8 + 16 + \ldots + 2^{\log_4 n}$$
$$= 2 + 4 + 8 + 16 + \ldots + \sqrt{n} = O(\sqrt{n})$$





# of regions in a 2d-tree with n points, which are intersected by a vertical straight line.



V(1)= 1

V(n)= 2+ 2V(n/4)

Summary

A 2d-tree for a set P of n points in the plane uses O(n) storage and can be built in O(n log n) time.

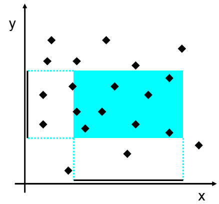
A rectangular range query on the 2d-tree takes $O(\sqrt{n} + k)$ time, where k is number of reported points.

Orthogonal Range Searching

1. Linear Range Search : 1-dim Range Trees
2. 2-dimensional Range Search : kd-trees
3. 2-dimensional Range Search : 2-dim Range Trees
4. Range Search in Higher Dimensions

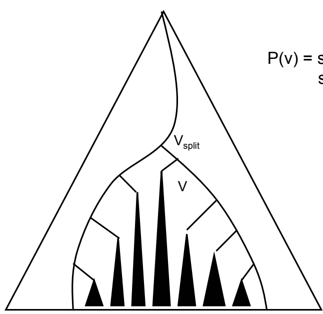
Range Trees

Two Dimensional Range Search



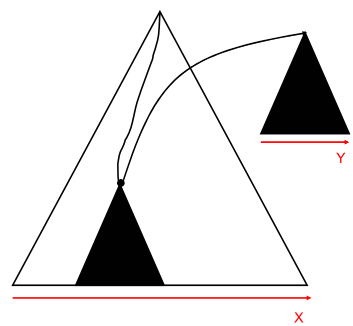
Assumption:
no two points have the same x or y coordinates

Canonical subset of a node



P(v) = set of points of the subtree with root v.

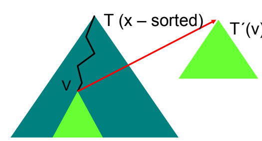
Associated tree



Range tree for a set P

1. The main tree (1st level tree) is a balanced binary search tree T built on the x – coordiante of the points in P.
2. For any internal or leaf node v in T, the canonical subset P(v) is stored in a balanced binary search tree $T_{assoc}(v)$ on the y – coordinate of the points. The node v stores a pointer to the root of $T_{assoc}(v)$ which is called the associated structure of v.

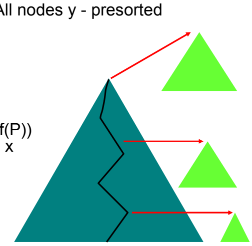
Constructing range trees



For all nodes in T store entire point information.
All nodes y - presorted

Build2DRangeTree(P)

1. Construct associated tree T´ for the points in P (based on y-coordinates)
2. If (|P| = 1) then return leaf(P), T´(leaf(P))
   else split P into P1, P2 via median x
   v1 = Build2DRangeTree(P1)
   v2 = Build2DRangeTree(P2)
   create node v, store x in v,
   left-child(v) =v1,right-child(v) = v2
   associate T´ with v



Lemma

Statement : A range tree on a set of n points in the plane requires O(n log n) storage.

Proof : A point p in P is stored only in the associated structure of nodes on the path in T towards the leaf containing p. Hence, for all nodes at a given depth of T, the point p is stored in exactly one associated structure. We know that 1 – dimensional range trees use linear storage, so associated structures of all nodes at any depth of T together use O(n) storage. The depth of T is O(log n). Hence total amount of storage required is O(n log n).

Search in 2-dim-range trees

Algorithm 2DRangeQuery(T,[x : x′] × [y : y′])

v_{split} = FindSplitNode(T,x,x′)
if (leaf(v_{split}) & v_{split} is in R) then report v, return
else v = left-child(v_{split})
  while not (leaf(v))
  do if ($x \le x_v$)
    then 1DRangeQuery(T_{assoc}(right-child(v)),[y : y′])
      v = left-child(v)
    else v = right-child(v)
  if (v is in R) then report v
  v = right-child(v_{split}) ... Similarly …

Analysis

Lemma: A query with an axis – parallel rectangle in a range tree storing n points takes $O(\log^2 n + k)$ time, where k is the number of reported points.

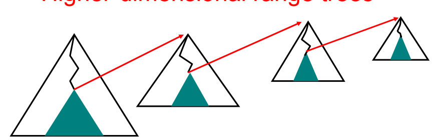
Proof : At each node v in the main tree T we spend constant time to decide where the search path continues and evt. call 1DRangeQuery. The time we spend in this recursive call is $O(\log n + k_v)$ where is k_v the number of points reported in this call. Hence the total time we spend is

$$\sum_v O(\log n + k_v)$$

Furthermore the search paths of x and x′in the main tree T have length O(log n). Hence we have

$$\sum_v O(\log n) = O(\log^2 n)$$

Higher-dimensional range trees



Time required for construction:

$T_2(n) = O(n \log n)$
$T_d(n) = O(n \log n) + O(\log n) * T_{d-1}(n)$
$\Rightarrow T_d(n) = O(n \log^{d-1} n)$

Time required for range query (without time to report points):

$Q_2(n) = O(\log^2 n)$
$Q_d(n) = O(\log n) + O(\log n) * Q_{d-1}(n)$
$\Rightarrow Q_d(n) = O(\log^d n)$

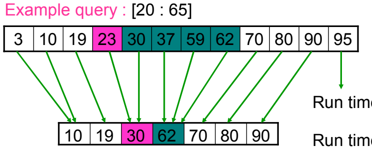
Search in Subsets

Given : Two ordered arrays A1 and A2.
key(A2) ⊂ key(A1)
query[x, x′]

Search : All elements e in A1 and A2 with x ≤ key(e) ≤ x′.

Idea : pointers between A1 and A2

Example query : [20 : 65]

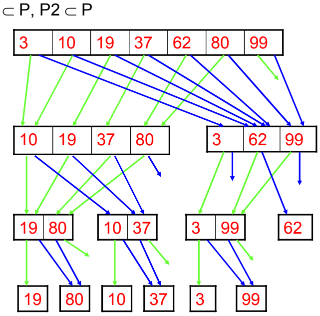


Run time : O(log n + k)

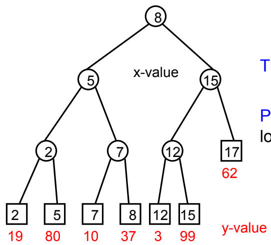
Run time : O(1 + k)

Fractional Cascading

Idea : P1 ⊂ P, P2 ⊂ P



Fractional Cascading



Theorem : query time can be reduced to O(log n + k).

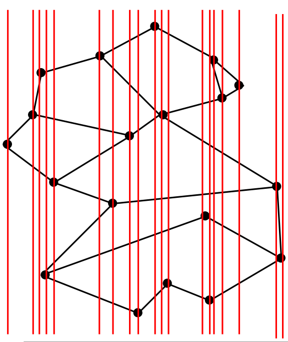
Proof : In d dimension a saving of one log n factor is possible.

Point Location

1. Trapezoidal decomposition.
2. A search structure.
3. Randomized, incremental algorithm for the construction of the trapezoidal decomposition.
4. Analysis.

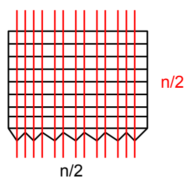
Point location in a map

Partition of the plane into slabs

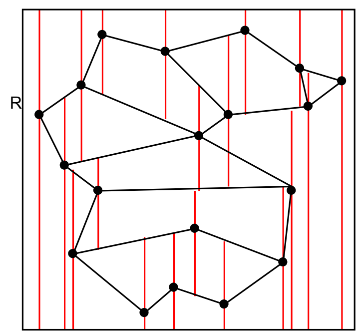


Query time : O(log n)
binary search in x and then binary search in y direction.

Storage space $O(n^2)$



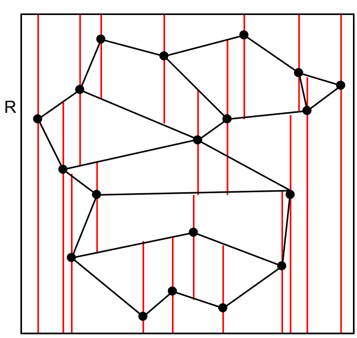
Partitioning into Trapezoids

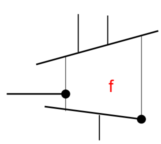


Assumption :
Segments are in „general position“

Observation :
Every vertical edge has one point in common with a segment end.

Observations



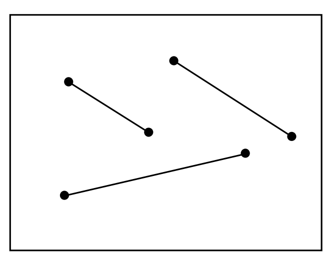


f is convex

f is bounded

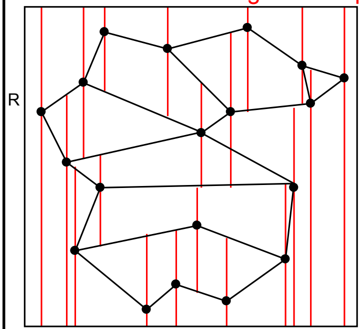
Every non - vertical side of f is part of a segment of S or an edge of R

Trapezoidal decomposition of set of line segments



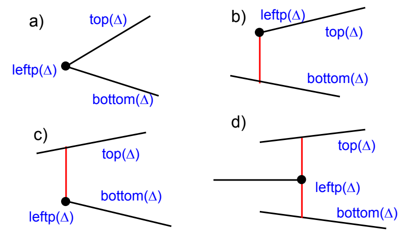
Lemma : Each face in a trapezoidal map of a set S of line segments in general position has 1 or 2 vertical sides and exactly two non-vertical sides

Left edge of a trapezoid



For every trapezoid $\Delta \in T(S)$, except the left most one, the left vertical edge of Δ is defined by a segment endpoint p, denoted by leftp(Δ) .

5 Cases (For left edge of a trapezoid)



e) It is left edge of R. This case occurs for a single trapezoid of T(S) only, namely the unique leftmost trapezoid of T(S)

Size of the trapezoidal map

Theorem: The trapezoidal map T(S) of a set of n line segments in general position contains at most 6n + 4 vertices and at most 3n + 1 trapezoids.

Proof (1): A vertex of T(S) is either

- a vertex of R or	4
- an endpoint of a segment in S or	2n
- a point where the vertical extension starting in an endpoint abuts on another segment or on the boundary R.	2 * (2n)
	6n + 4

Size of the trapezoidal map

Theorem: The trapezoidal map T(S) of a set of n line segments in general position contains at most 6n + 4 vertices and at most 3n + 1 trapezoids.

Proof (2): Each trapezoid has a unique point leftp(Δ), which is

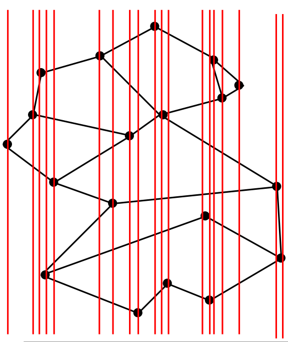
- the lower left corner of R	1
- the left endpoint of a segment (can be leftp(Δ) of at most two different trapezoids)	2n
- the right endpoint of a segment (can be leftp(Δ) of atmost one trapezoid)	n
	3n + 1

Point Location

1. Trapezoidal decomposition.
2. A search structure.
3. Randomized, incremental algorithm for the construction of the trapezoidal decomposition.
4. Analysis.

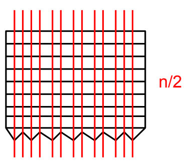
Point location in a map

Partition of the plane into slabs

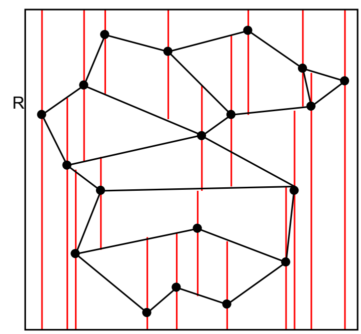


Query time : O(log n)
binary search in x and then binary search in y direction.

Storage space $O(n^2)$



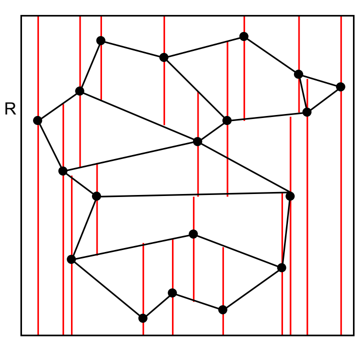
Partitioning into Trapezoids

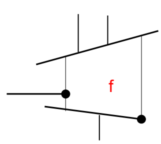


Assumption :
Segments are in „general position“

Observation :
Every vertical edge has one point in common with a segment end.

Observations



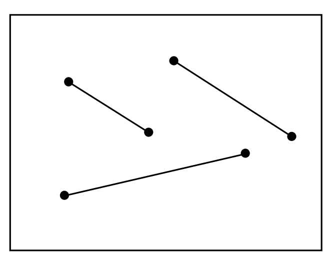


f is convex

f is bounded

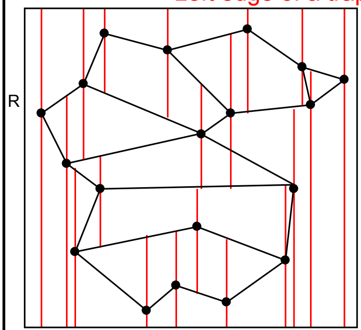
Every non - vertical side of f is part of a segment of S or an edge of R

Trapezoidal decomposition of set of line segments



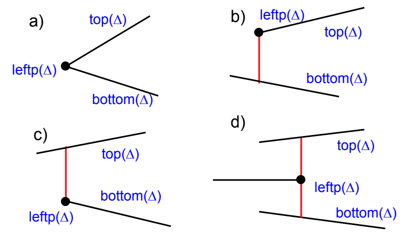
Lemma : Each face in a trapezoidal map of a set S of line segments in general position has 1 or 2 vertical sides and exactly two non-vertical sides

Left edge of a trapezoid



For every trapezoid $\Delta \in T(S)$, except the left most one, the left vertical edge of Δ is defined by a segment endpoint p, denoted by leftp(Δ) .

5 Cases (For left edge of a trapezoid)



e) It is left edge of R. This case occurs for a single trapezoid of T(S) only, namely the unique leftmost trapezoid of T(S)

Size of the trapezoidal map

Theorem: The trapezoidal map T(S) of a set of n line segments in general position contains at most 6n + 4 vertices and at most 3n + 1 trapezoids.

Proof (1): A vertex of T(S) is either

- a vertex of R or	4
- an endpoint of a segment in S or	2n
- a point where the vertical extension starting in an endpoint abuts on another segment or on the boundary R.	2 * (2n)
	6n + 4

Size of the trapezoidal map

Theorem: The trapezoidal map T(S) of a set of n line segments in general position contains at most 6n + 4 vertices and at most 3n + 1 trapezoids.

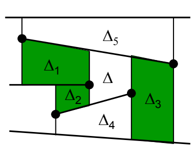
Proof (2): Each trapezoid has a unique point leftp(Δ), which is

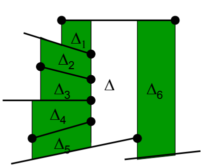
- the lower left corner of R	1
- the left endpoint of a segment (can be leftp(Δ) of at most two different trapezoids)	2n
- the right endpoint of a segment (can be leftp(Δ) of atmost one trapezoid)	n
	3n + 1

Adjacent trapezoids

Two trapezoids Δ and Δ' are adjacent if they meet along a vertical edge.

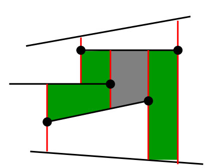
1) Segments in general position : A trapezoid has atmost four adjacent trapezoids





2) Segments not in general position: A trapezoid can have an arbitrary number of adjacent trapezoids.

Vertical neighbors:Upper, lower left neighbor



Trapezoid Δ' is (vertical) neighbor of Δ

top(Δ) = top(Δ') or bottom(Δ) = bottom(Δ')

In the first case Δ' is upper left neighbor of Δ, in the second case Δ' is lower left neighbor of Δ.

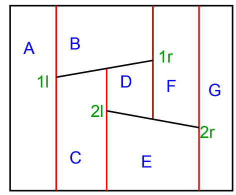
Representing trapezoidal maps

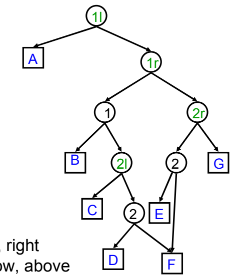
There are records for all line segments and endpoints of S, the structure contains records for trapezoids of T(S), but not for vertices or edges of T(S).

The record for trapezoid Δ stores pointers to top(Δ), and bottom(Δ) , pointers to leftp(Δ) and rightp(Δ) and finally pointers to its atmost 4 neighbors.

Δ is uniquely defined by top(Δ), bottom(Δ), leftp(Δ) and rightp(Δ).

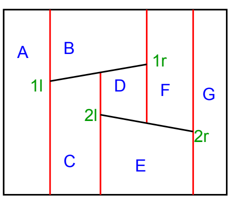
A search structure





End points decide between left, right
Segments decide between below, above

Example : Search structure



A randomized incremental algorithm

Input : A set S of n non-crossing line segments
Output : The trapezoidal map T(S) and a search structure D(S) for T(S) in a bounding box.

Determine a bounding box R, initialize T and D

Compute a random permutation $s_1, s_2, \dots, s_n$ of the elements of S

for i = 1 to n
do add s_i and change $T(S_{i-1})$ into $T(S_i)$ and $D(S_{i-1})$ into $D(S_i)$

Invariant :
In the step i $T(S_i)$ is correct trapezoidal map of S_i and $D(S_i)$ is an associated search structure.

A randomized incremental algorithm

Input : A set of n non-crossing line segments
Output : The trapezoidal map T(S) and a search structure D for T(S) in a bounding box.

Determine a bounding box R, initialize T and D

Compute a random permutation $s_1, s_2, \dots, s_n$ of the elements of S

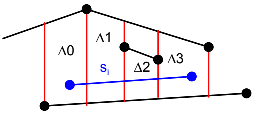
for i = 1 to n
do Find the set $\Delta_0, \Delta_1, \dots, \Delta_k$ of trapezoids in T properly intersected by s_i.
Remove $\Delta_0, \Delta_1, \dots, \Delta_k$ from T and replace them by new trapezoids that appear because of the intersection of s_i.
Remove the leaves for $\Delta_0, \Delta_1, \dots, \Delta_k$ from D and create leaves for the new Trapezoids.
Link the new leaves to the existing inner nodes by adding some new inner nodes.

Questions

How can we find the intersecting trapezoids?

How can T and D be updated
a) if new segment intersects no previous trapezoid
b) if new segment intersects previous trapezoids

Finding the intersecting trapezoids



Δ_{j+1} is lower right neighbour of Δ_j

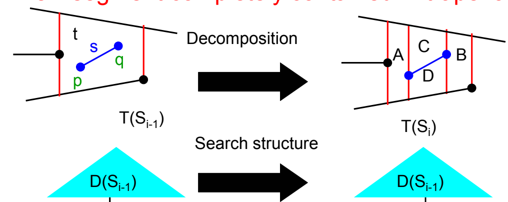
In $T(S_i)$ exactly those trapezoids are changed, which are intersected by s_i

if rightp(Δ_j) lies above s_i
 then Let Δ_{j+1} be the lower right neighbor of Δ_j.
 else Let Δ_{j+1} be the upper right neighbor of Δ_j

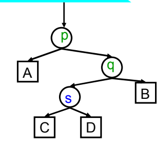
Clue :

Δ_0 can be found by a query in the search structure $D(S_{i-1})$ constructed in iteration stage i -1.

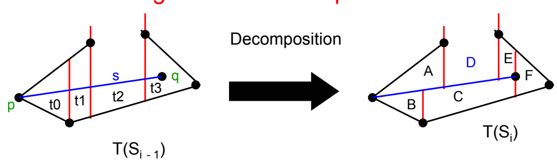
New segment completely contained in trapezoid



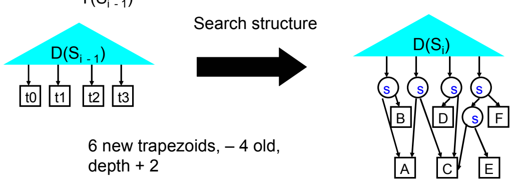
4 new trapezoids, -1 old trapezoid, search depth + 3



New segment intersects previous ones



Search structure



6 new trapezoids, – 4 old, depth + 2

Estimation of the depth of the search structure

Let S be a set of n segments in general position, q be an arbitrary fixed query point.

Depth of D(S):
 worst case : 3n,
 average case : O(log n)

Consider the path traversed by the query for q in D

Let X_i = # of nodes on the search path for q created in iteration step i.

X_i <= 3

P_i = probability that there exists node on the search path for q that is created in iteration step i.

$E[X_i]$ <= 3 P_i

Observation

Iteration step i contributes a node to the search path for q exactly if $\Delta_q(S_{i-1})$, the trapezoid containing q in $T(S_{i-1})$, is not the same as $\Delta_q(S_i)$, the trapezoid containing q in $T(S_i)$

$$P_i = Pr[\Delta_q(S_i) \neq \Delta_q(S_{i-1})].$$

If $\Delta_q(S_i)$ is not same as $\Delta_q(S_{i-1})$, then $\Delta_q(S_i)$ must be one of the trapezoids created in iteration i.

$\Delta_q(S_i)$ does not depend on the order in which the segments in S_i have been inserted.

Backwards analysis :

We consider $T(S_i)$ and look at the probability that $\Delta_q(S_i)$ disappears from the trapezoidal map when we remove the segment s_i.

$\Delta_q(S_i)$ disappears if and only if one of top($\Delta_q(S_i)$), bottom($\Delta_q(S_i)$), leftp($\Delta_q(S_i)$), or right($\Delta_q(S_i)$) disappears with removal of s_i .

Prob[top($\Delta_q(S_i)$)] = Prob[bottom($\Delta_q(S_i)$)] = 1/i.

Prob[leftp($\Delta_q(S_i)$)] disappears is at most 1/i.

Prob[rightp($\Delta_q(S_i)$)] disappears is at most 1/i.

$P_i = Pr[\Delta_q(S_i) \neq \Delta_q(S_{i-1})] = Pr[\Delta_q(S_i) \notin T(S_{i-1})]$
 <= 4/i

$$E\left[\sum_{i=1}^{n} Xi\right] \leq \sum_{i=1}^{n} 3P_i \leq \sum_{i=1}^{n} \frac{12}{i} = 12\sum_{i=1}^{n}\frac{1}{i} = 12H_n = O(\log n)$$

Analysis of the size of search structure

Leaves in D are in one – to – one correspondence with the trapezoids in Δ, of which there are O(n).

The total number of nodes is bounded by :

$$O(n)+\sum_{i=1}^{n} \text{(\# of inner nodes created in iteration step i)}$$

The worst case upper bound on the size of the structure

$$O(n)+\sum_{i=1}^{n} O(i) = O(n^2)$$

Analysis of the size of search stucture

Theorem: The expected number of nodes of D is O(n).

Proof: The # of leaves is in O(n). Consider the internal nodes:
X_i = # of internal nodes created in iteration step i

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

$$\delta(\Delta, s) := \begin{cases} 1 & \text{if } \Delta \text{ disappears from } T(S_i) \text{ when s is removed from } S_i \\ 0 & \text{otherwise} \end{cases}$$

There are at most four segments that cause a given trapezoid to disappear

$$\sum_{s\in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \le 4|T(S_i)| = O(i)$$

$$E[k_i] = \frac{1}{i}\sum_{s\in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \le \frac{O(i)}{i} = O(1)$$

The expected number of newly created trapezoids is O(1) in every iteration of the algorithm, from which the O(n) bound on the expected amount of storage follows.

$$\Rightarrow E\left[\sum_{i=1}^{n} X_i\right] = O(n)$$

Summary

Let S be a planar subdivision with n edges. In O(n log n) expected time one can construct a data structure that uses O(n) expected storage, such that for any query point q, the expected time for a point location query is O(log n).

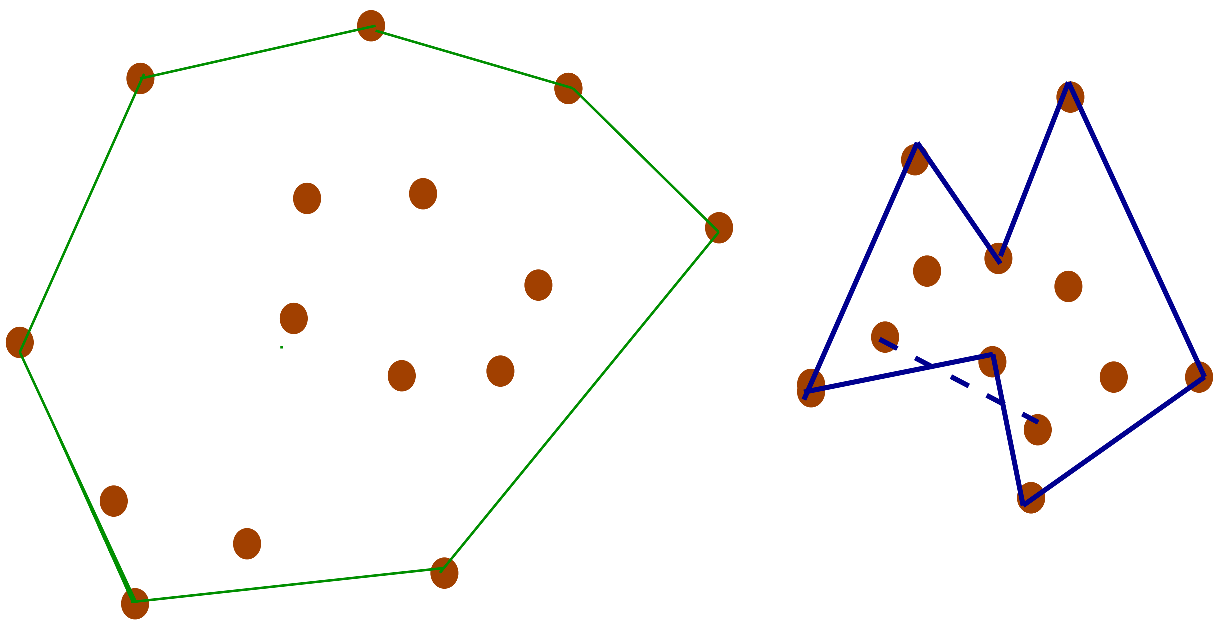
An overview of Lecture 7.1

- Definitions: Convex set, Extreme point, Convex Hull
- Lower Bound
- Point Pruning
- Edge Pruning
- Jarvis March
- Graham's Scan
- Summary

Definitions: Convex set, Extreme point

- A set $S \subseteq E^2$ is convex iff for every $p1,\ p_2 \in S$, the segment $p_1 p_2$ is completely within S.

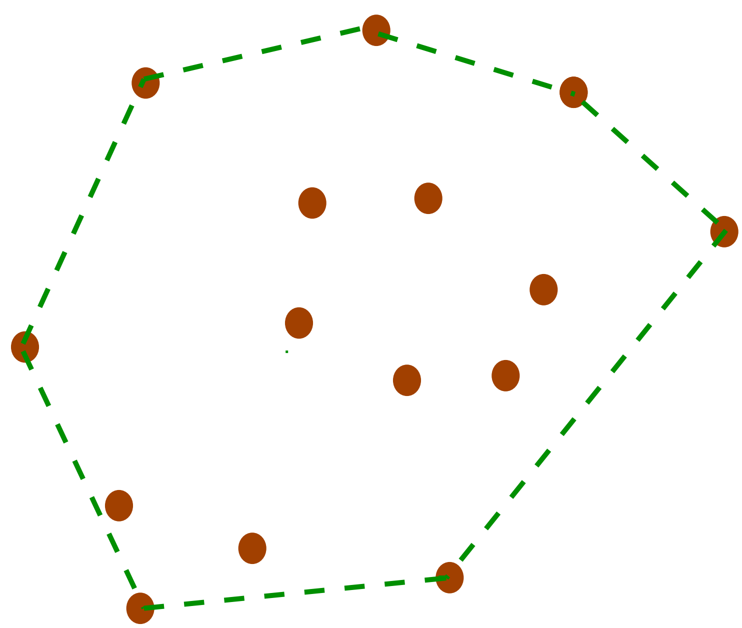
- A point p in a convex set S is said to be extreme iff there is no segment $ab \subseteq S$ with p in its interior.



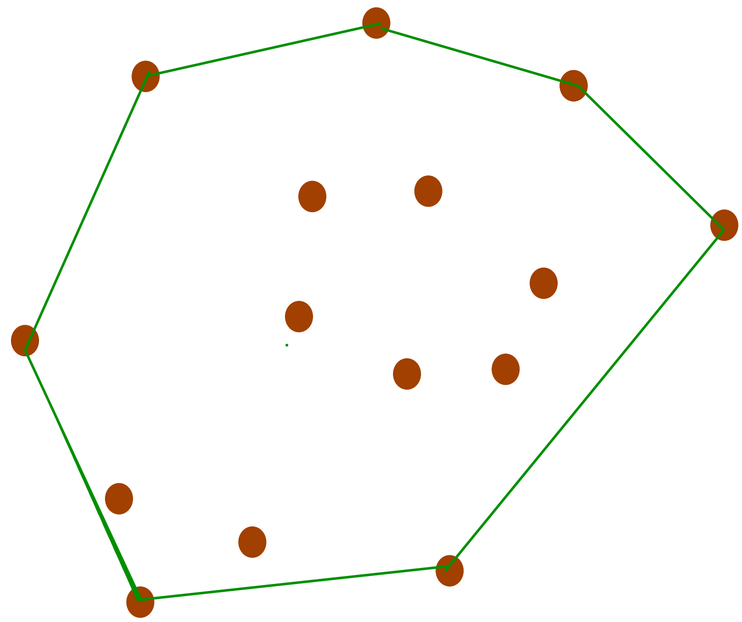
Problem Formulation

Given: A set P of n points in the plane.

Find: Smallest convex set containing P. It is called the *convex hull* of P, and is denoted by $CH(P)$.



Problem Formulation



- Since P is finite, the boundary of $CH(P)$ is a simple polygon with a subset of points of P as its extreme points(corners).

- $CH(P)$ is considered determined once its extreme points, ordered around the boundary are found.

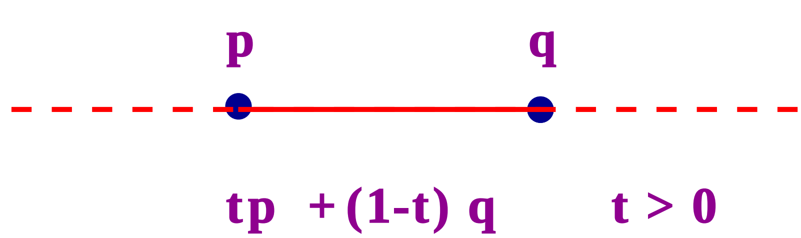
- Simplifying assumption: No pair of points has the same x- or y-coordinate.

Equivalent Definitions of Convex Hull

- A *convex combination* of points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$ is a point $q = \alpha_1(x_1, y_1) + \alpha_2(x_2, y_2) + \ldots + \alpha_n(x_n, y_n)$ with $\alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i = 1$.

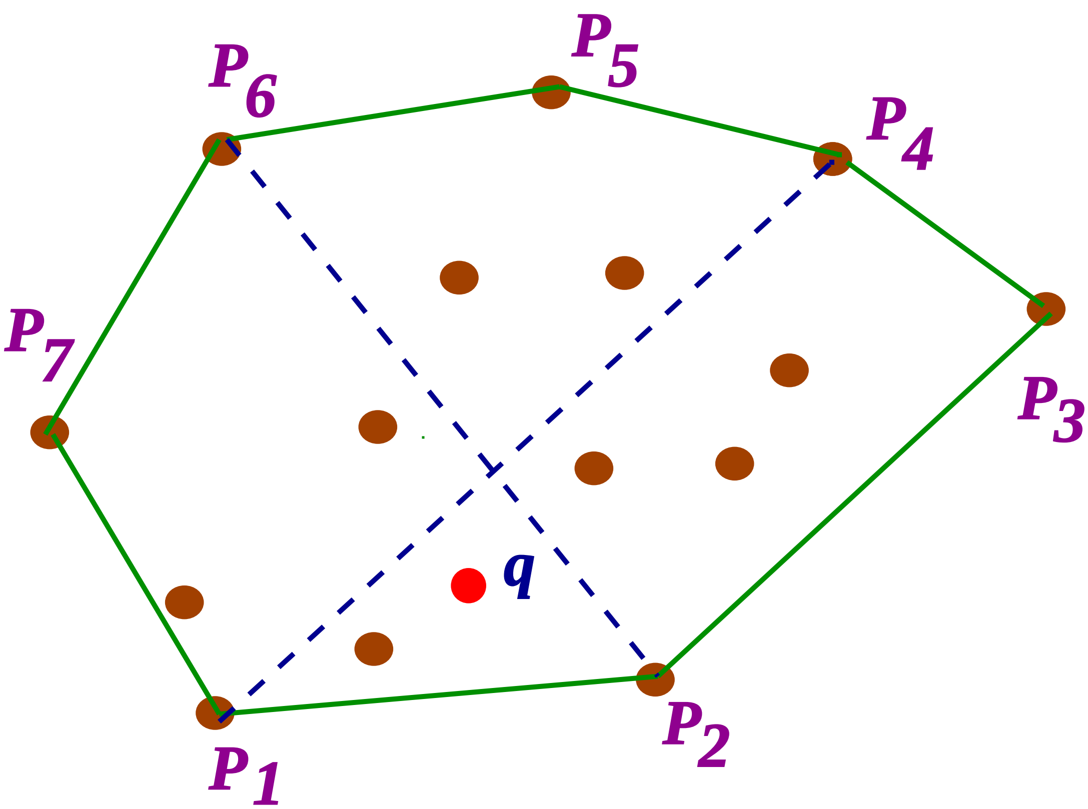
In other words,
$q = \sum_{i=1}^{n} \alpha_i p_i$ with $\alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i = 1$.

Example: Convex combination of two points p and q.



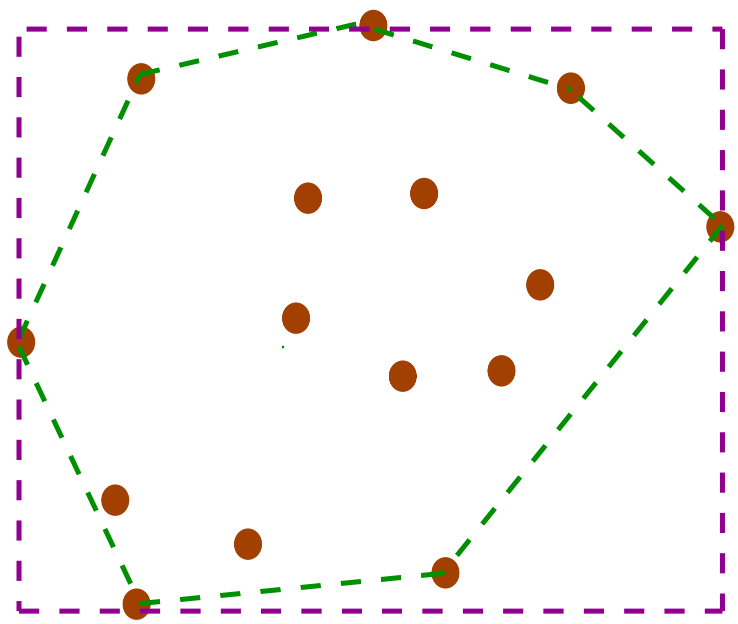
Equivalent Definitions of Convex Hull

- Let P be a set of n points. A point q in $CH(P)$ is the convex combination of its extreme points



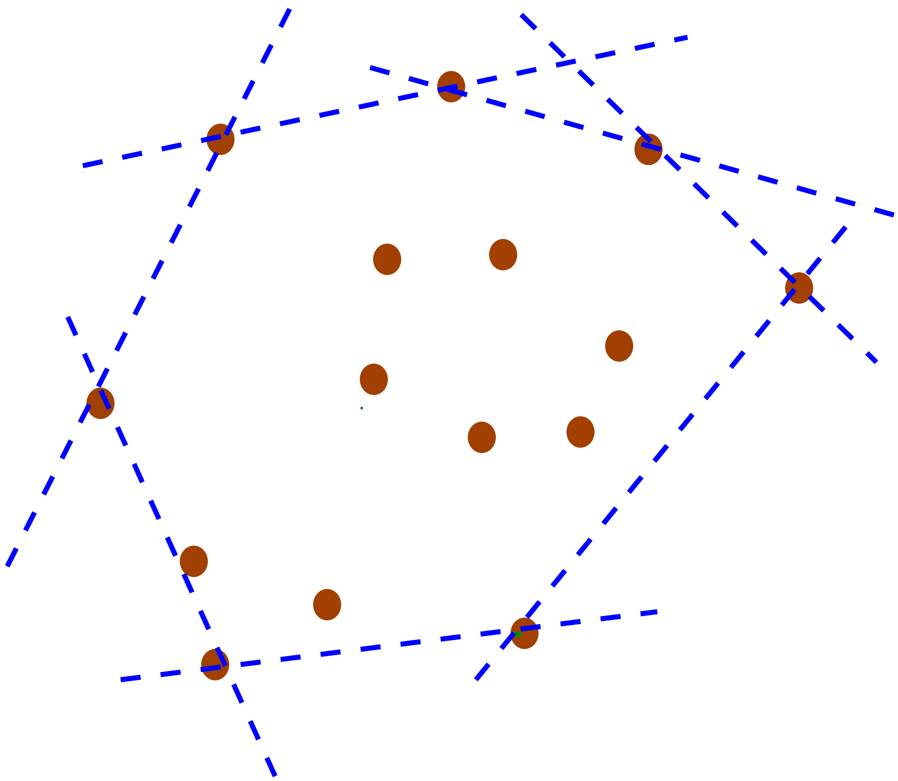
Equivalent Definitions of Convex Hull

- intersection of all convex sets containing P.



Equivalent Definitions of Convex Hulls

- intersection of all half-planes containing P.



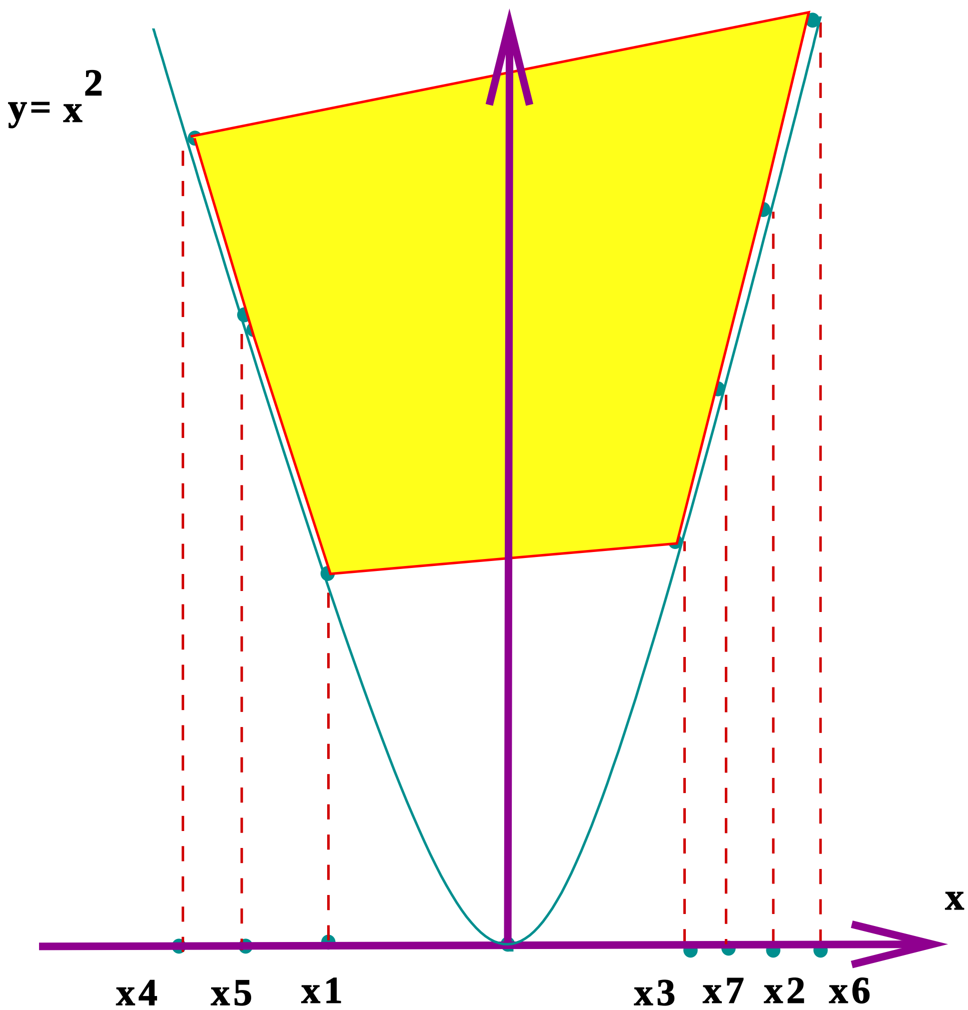
$$a_1 x + a_2 y = b$$

$$a_1 x + a_2 y \geq b$$

$$a_1 x + a_2 y \leq b$$

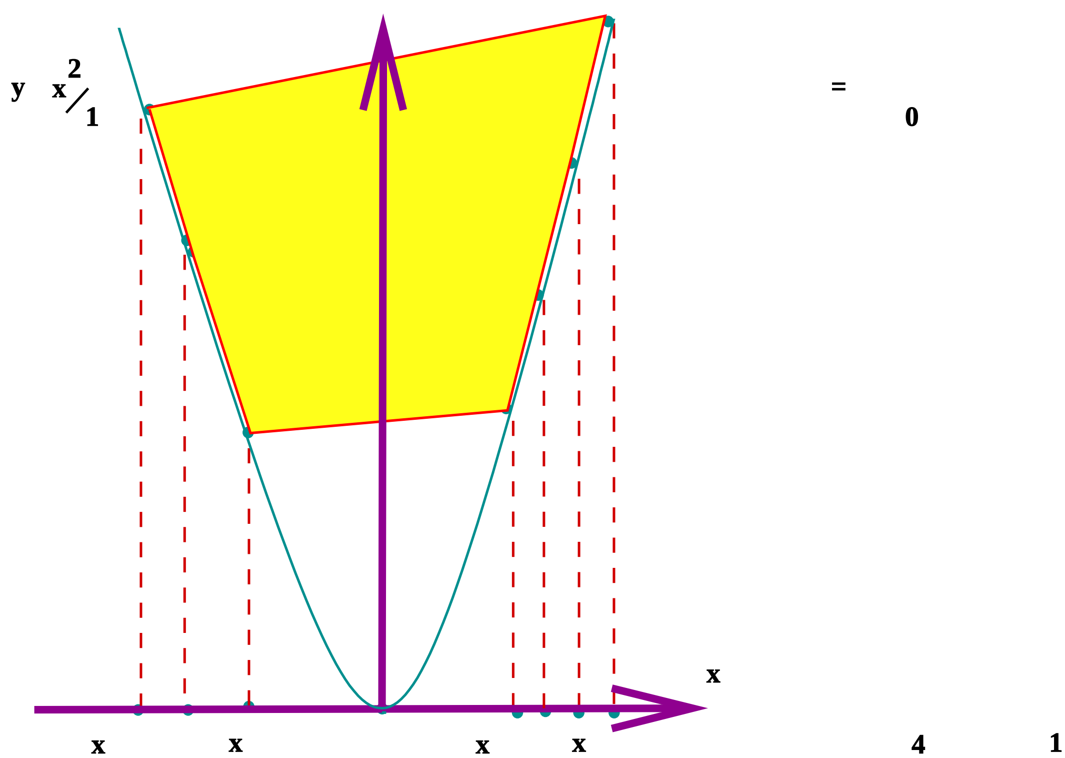
Lower Bound

- Sorting of real numbers can be transformed in linear time into the convex hull problem.
- Transformation: $x_i \rightarrow (x_i, x_i^2)$



Lower Bound

- Enumerating the extreme points around the convex hull is equivalent to sorting the points $x_1, x_2, x_3, x_4, x_5, x_6$.

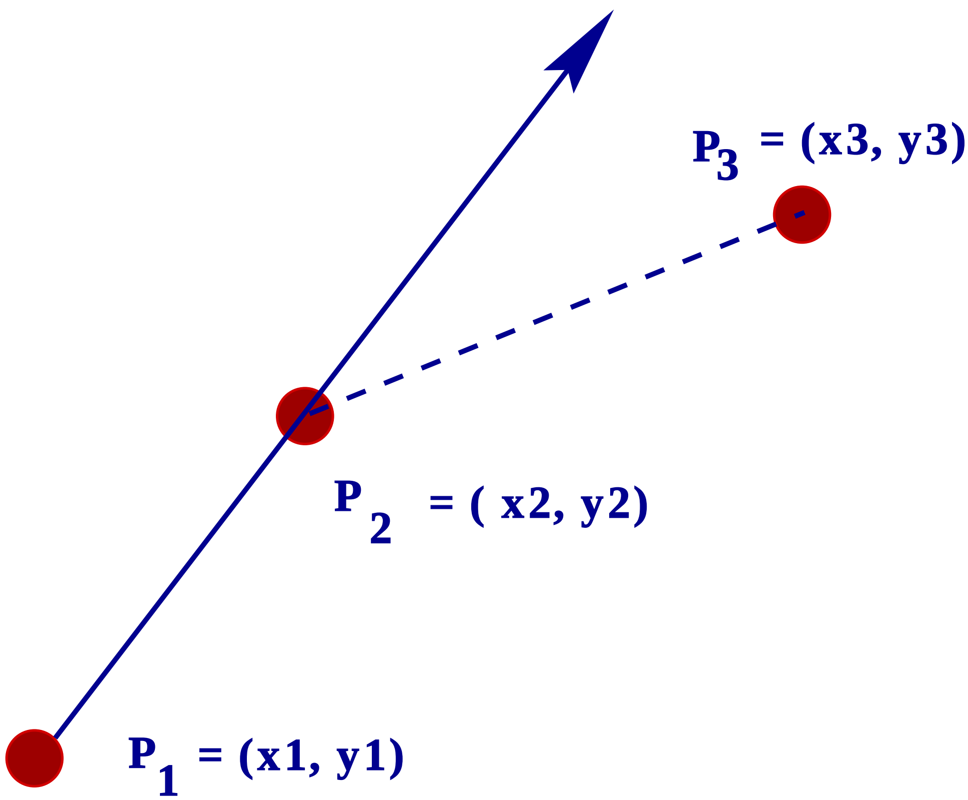


- Sorting requires $\Omega(n \log n)$ time. Hence, Convex hull problem must have the same lower bound.

Left and Right Turns

- A sequence $\{p_1,\ p_2,\ p_3\}$ of points makes a right turn at p_2 iff p_3 is to the right or on the line through p_1 and p_2(when looking from p_1 towards p_2).

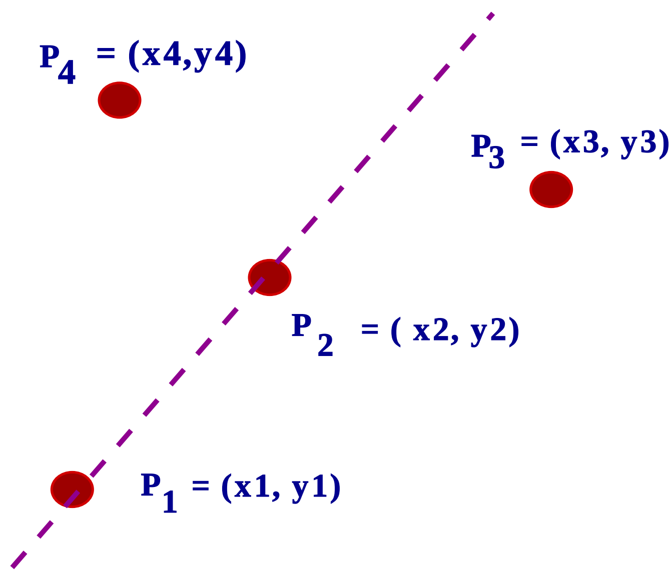
- Otherwise $\{p_1,\ p_2,\ p_3\}$ makes a left turn at p_2



Left and Right Turns

- Consider $\Delta p_1 p_2 p_3$. The double of its area (disregarding the sign) is

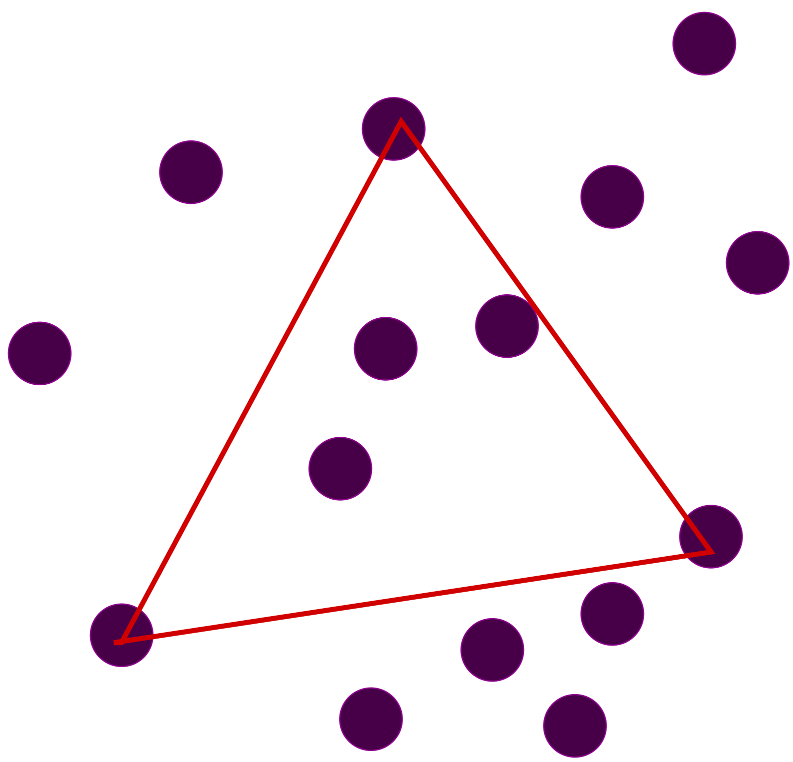
$$\left\| \begin{array}{ccc} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{array} \right\|$$



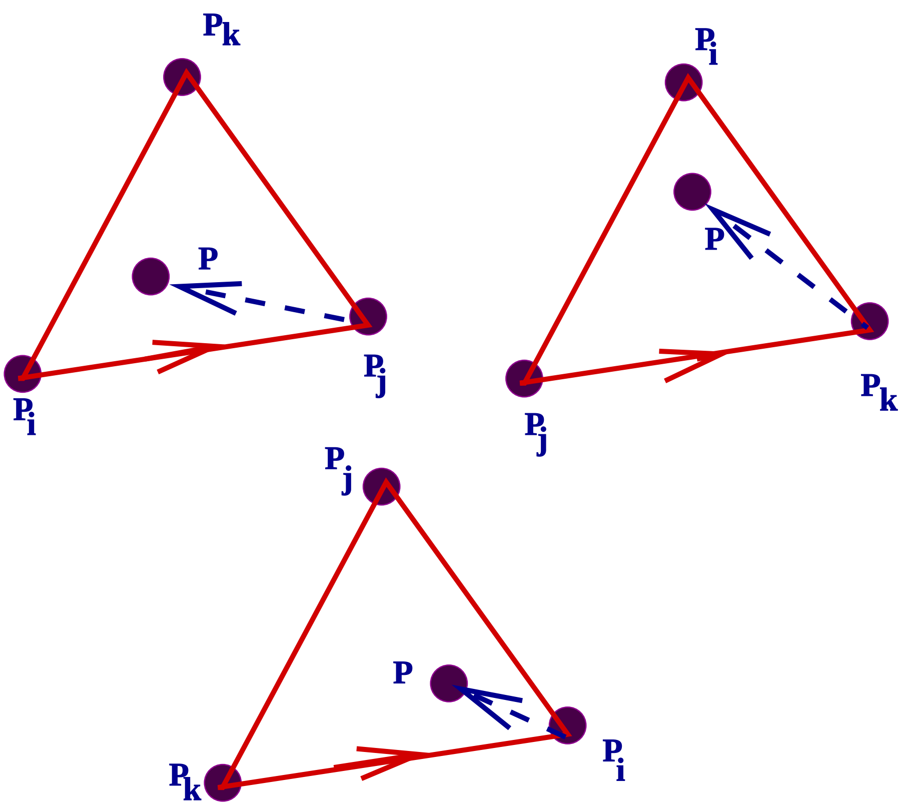
- The sign is $+$ iff $\{p_1,\ p_2,\ p_3\}$ appears in the counterclockwise order on $\Delta p_1 p_2 p_3$. Hence, $\{p_1,\ p_2,\ p_3\}$ turns left at p_2.

Point Pruning

- A point $p \in P$ is not extreme in $CH(P)$ iff $\exists\{p_i,\ p_j,\ p_k\} \in P - \{p\} : p \in \Delta p_i p_j p_k$



Finding all Extreme Points

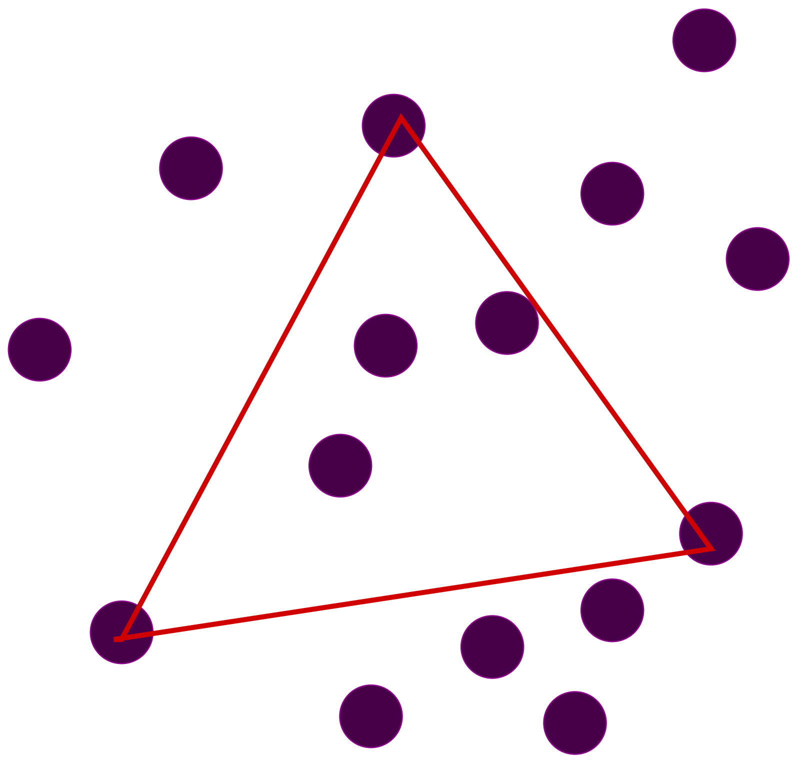


- $p \in \Delta p_i p_j p_k$ can be verified in $O(1)$ time; $\{p_i, p_j, p\}$, $\{p_j, p_k, p\}$ and $\{p_k, p_i, p\}$ are all left turns if we traverse $\Delta p_i p_j p_k$ in the anticlockwise direction.

If $p \in \Delta p_i p_j p_k$
then
eliminate p

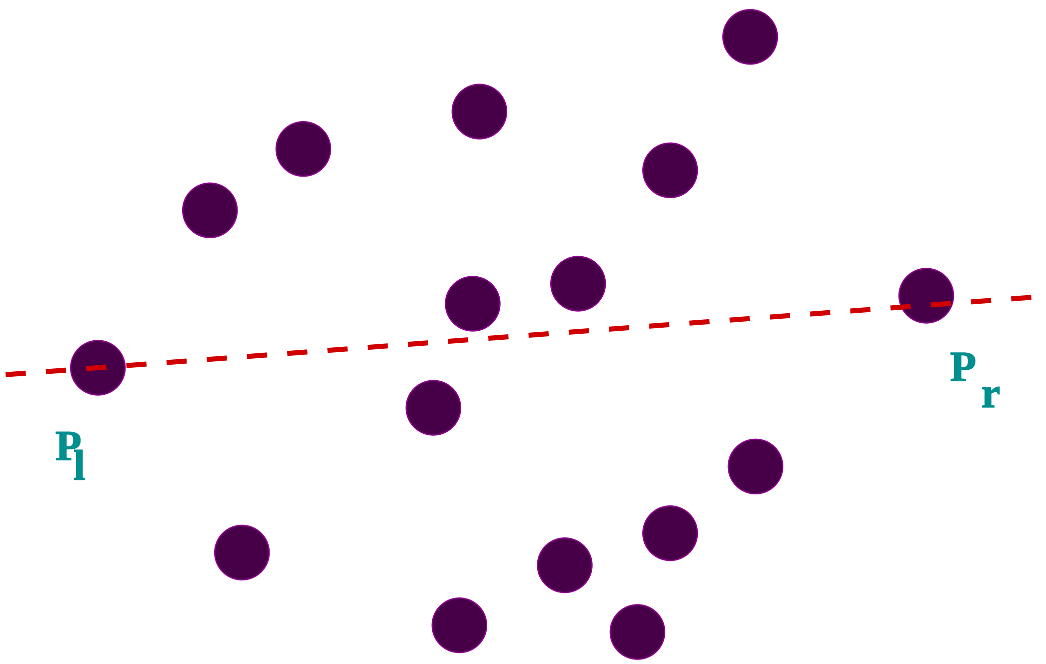
Algorithm-Point Pruning

Algorithm: For each triangle, we test in $O(n)$ time whether all the remaining points are inside or outside the triangle.



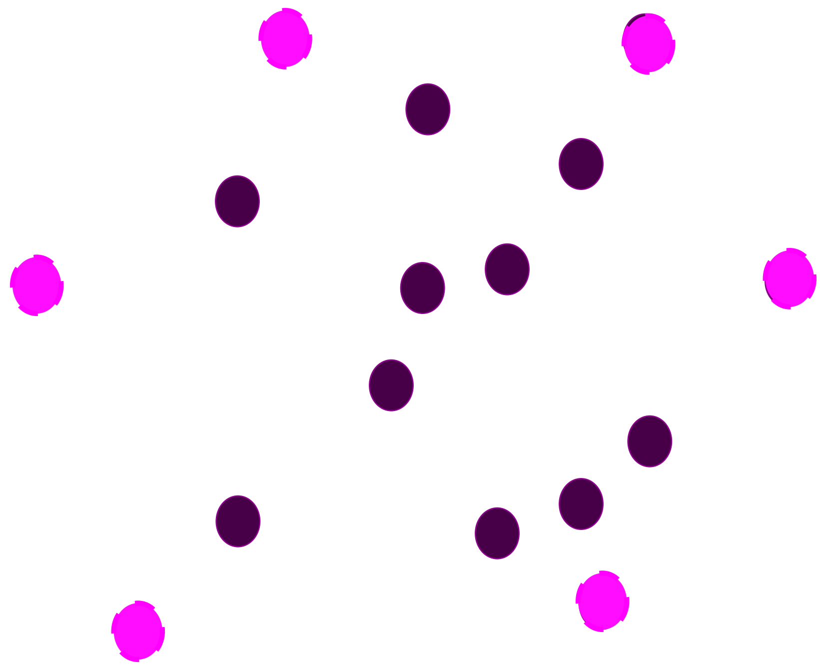
- In the worst-case, there are $O(n^3)$ triangles to consider.
- Overall complexity: $O(n^4)$

Improved Point Pruning



- Can be improved to $O(n^2)$ by fixing p_i and p_j to the leftmost point p_l and rightmost point p_r. Both p_l and p_r are extreme points and can be found in $O(n)$ time.

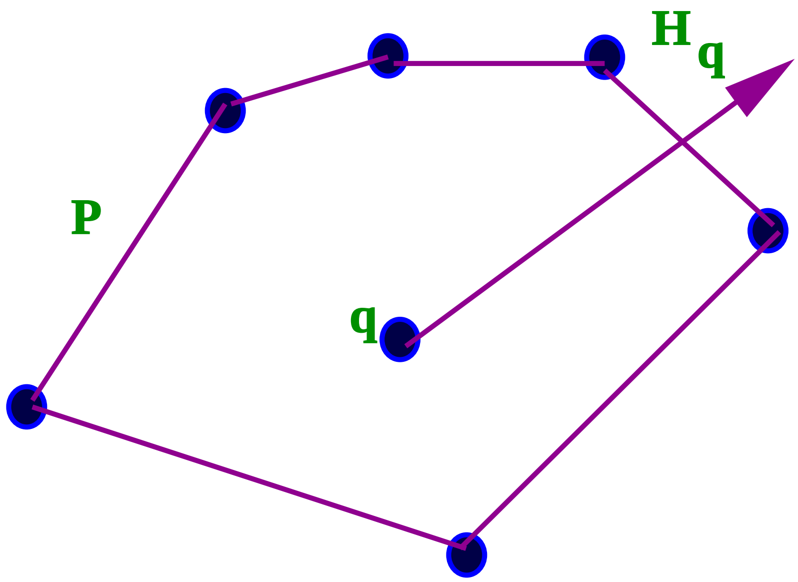
Sorting of Extreme Points



- It remains to sort the extreme points.

Sorting of Extreme Points

Method1

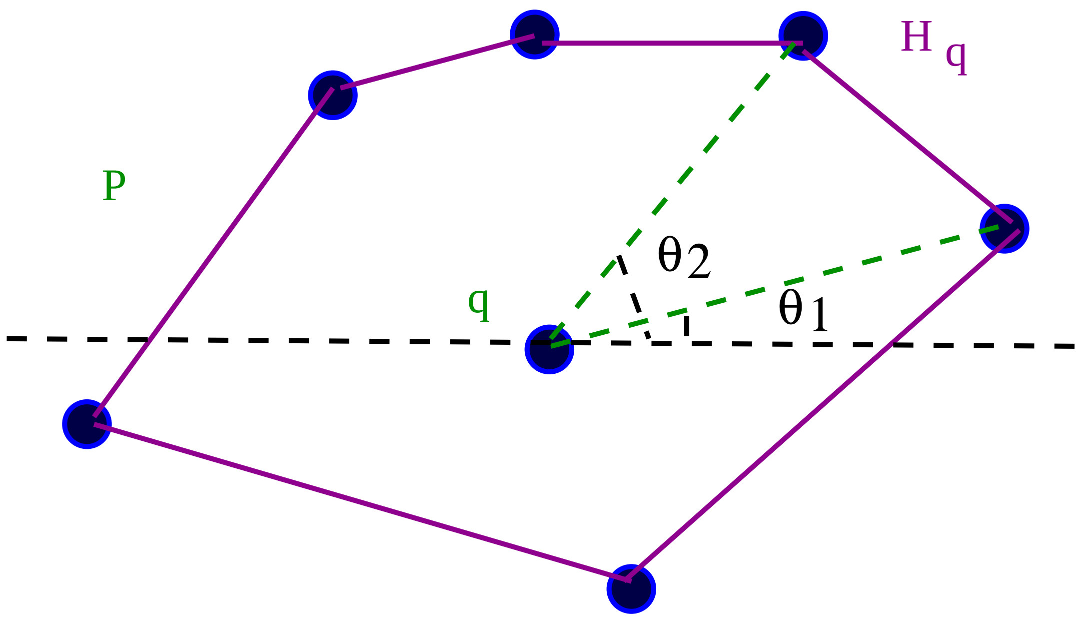


- H_q: half-line rooted at a point q in the interior of a convex set S.

- H_q intersects the boundary of S in exactly one place(for all possible directions).

Sorting of Extreme Points

- Sort extreme points of P in increasing order of their polar angles around a point q known to be in the interior of $CH(P)$. Requires $O(nlogn)$ time.



- Interior point: centroid of the extreme points:

$$(q_x, q_y) = (\sum_{i=1}^{n} x_i/n, \sum_{i=1}^{n} y_i/n)$$

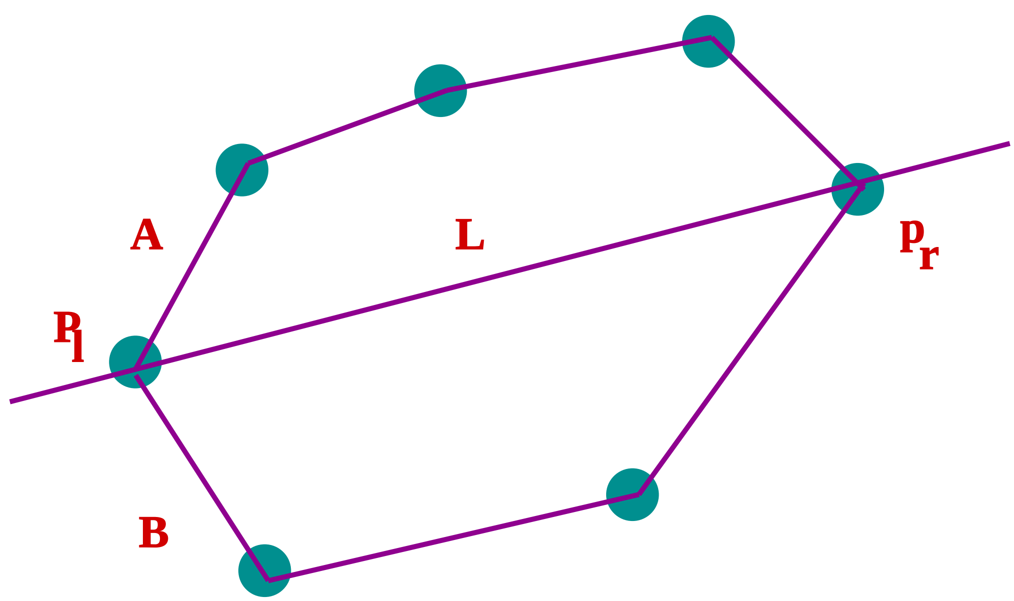
where $p_i = (x_i, y_i)$. Requires $O(n)$ time.

Sorting of Extreme Points

Method2

- Draw a line L through p_l and p_r.
- Partition the remaining extreme points into two groups:

A: extreme points above L.
B: extreme points below L.

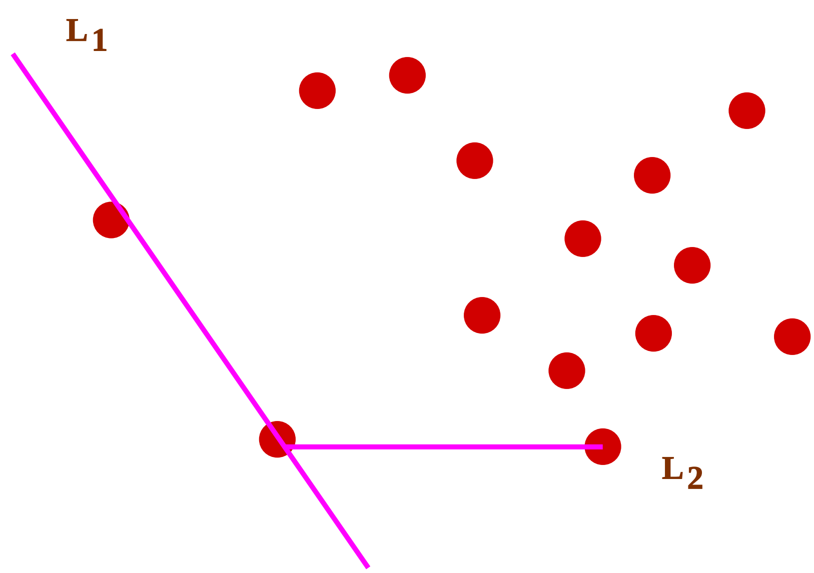


- Sort A by decreasing x-coordinate.
- Sort B by increasing x-coordinate.
- All this can be done in $O(nlogn)$ time.

Edge Pruning

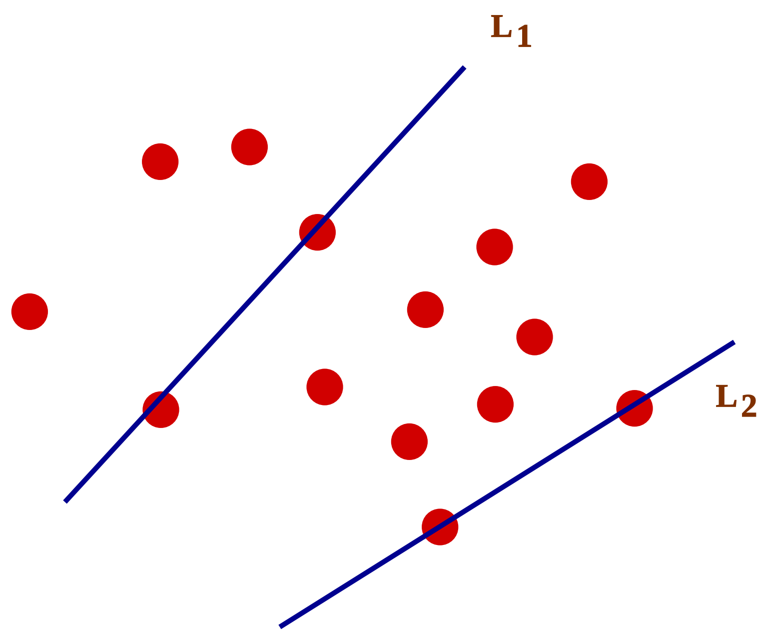
- General idea: Identify boundary edges rather than extreme points.

- A segment between two points of P is a boundary edge iff all remaining points of P are on one side of the line through the segment.



Edge Pruning

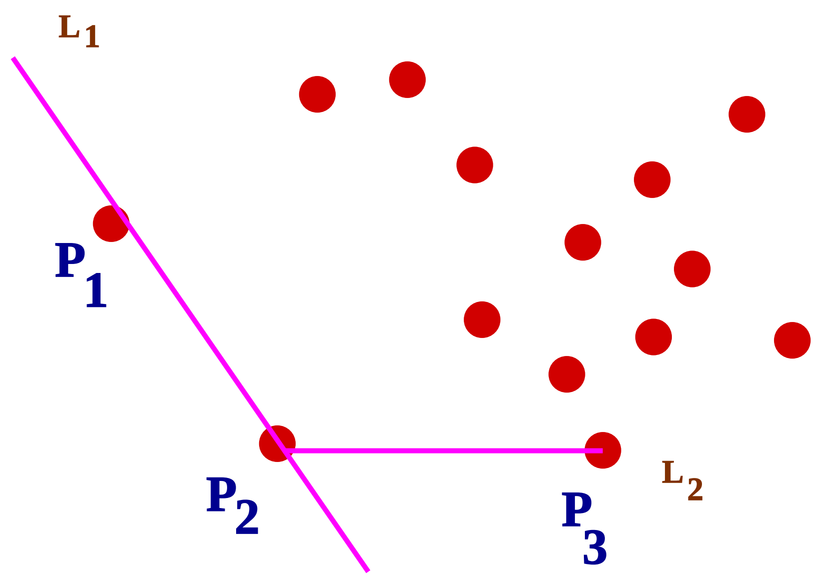
- Algorithm: For each pair of P-points p_i and p_j, check in $O(n)$ time if all the remaining points of P are on the same side of the line through p_i and p_j.
- Number of pairs is $O(n^2)$. All boundary edges can be identified in $O(n^3)$ time.



- End points of boundary edges are extreme points. They need to be sorted. This can be done in $O(nlogn)$ time.

Jarvis's March(1973)

- Can we improve the $O(n^3)$ edge pruning algorithm?

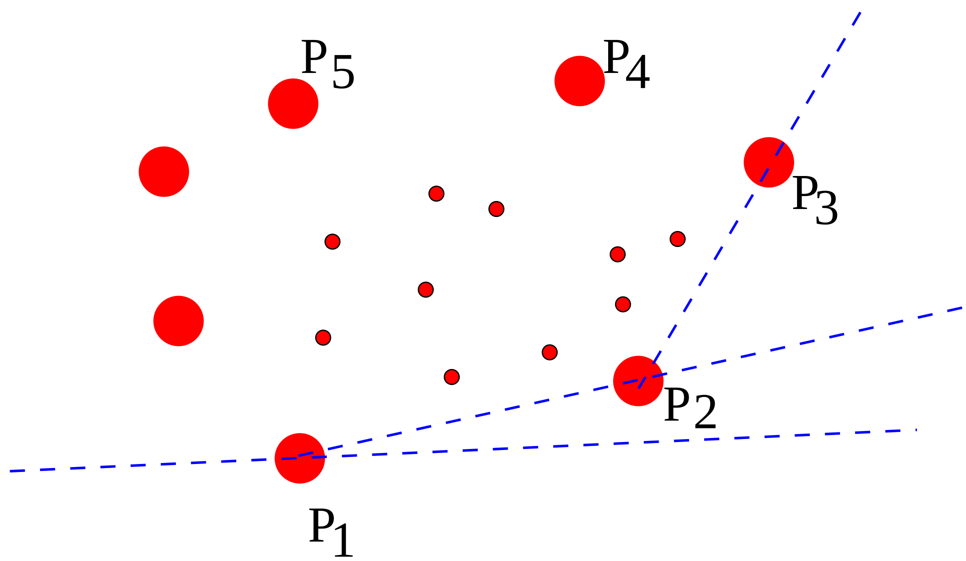


<u>Observation</u>

- When a boundary edge $p_i p_j$ has been identified there must exist another boundary edge with p_j as one of its endpoints.

Jarvis's March

General idea: use one extreme edge as an anchor for finding the next.



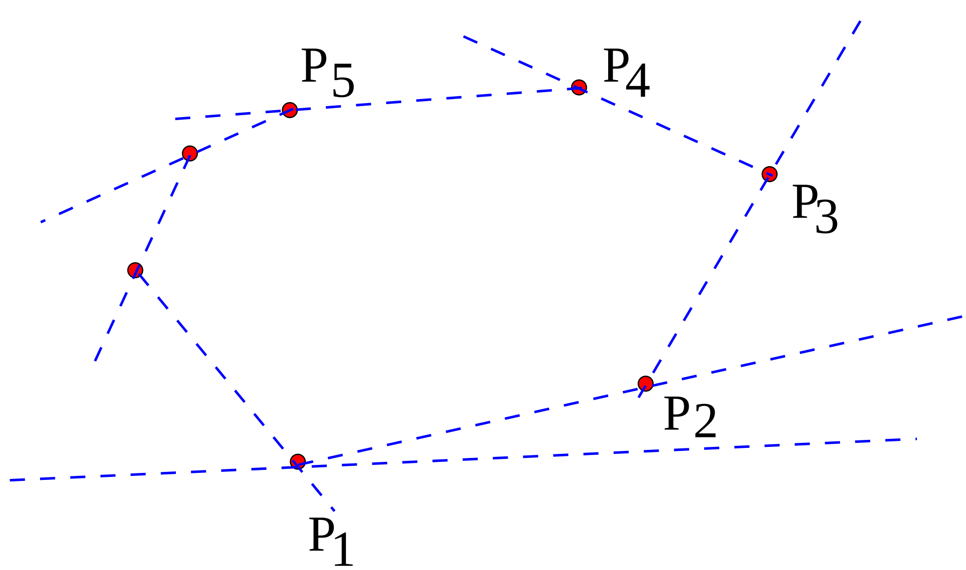
- The algorithm output the extreme points in the order in which they occur around the hull boundary.

Jarvis's march is also known as

gift wrapping method

Jarvis's March - Continued

- Find the point p_1 with lowest y-coordinate.
- Find the point p_2 such that its polar angle with p_1 as origion is smallest possible.
- Find the point p_3 such that its polar angle with p_2 as origion is smallest possible.

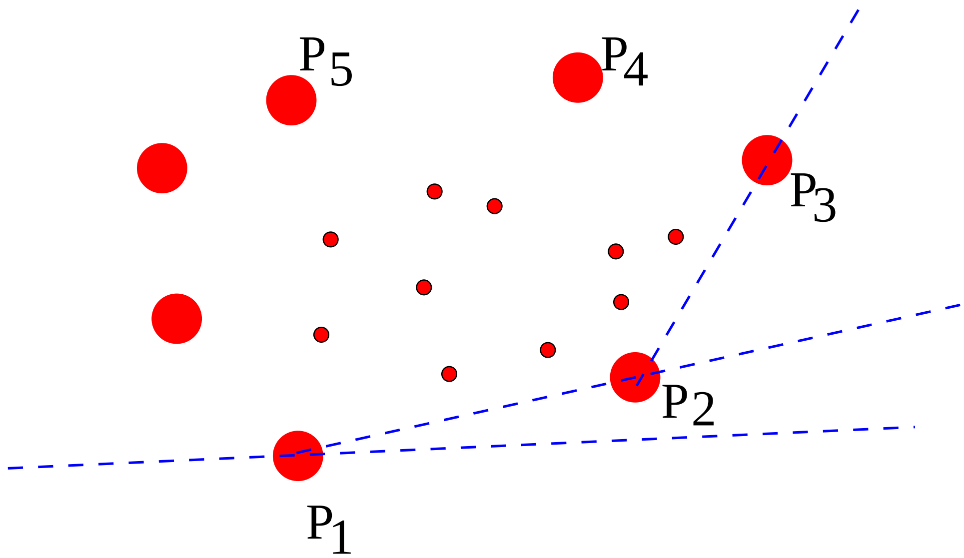


- Continue until the point of P with highest y-coordinate has been identified.

" Turn around " and repeat.

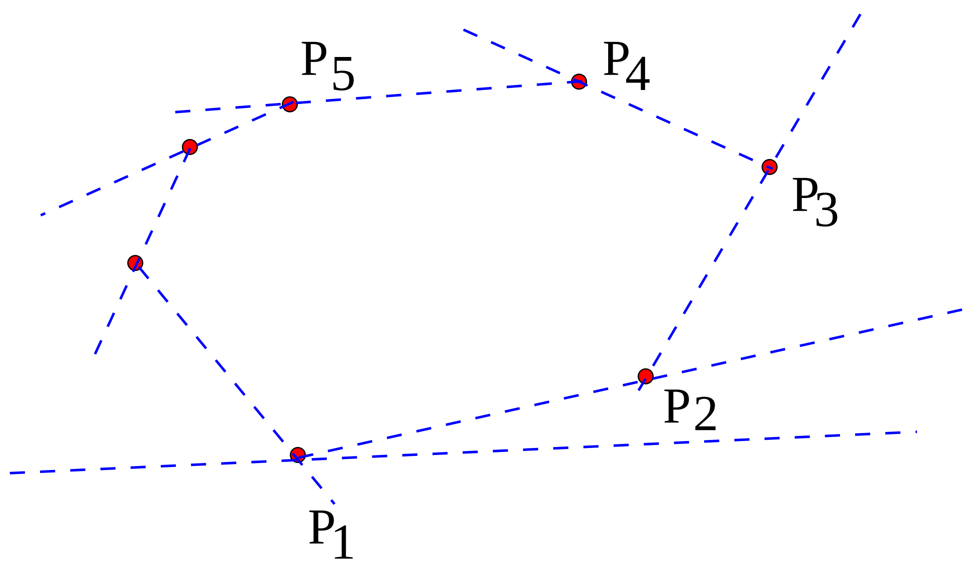
Time Complexity of Jarvis's March

- To find the points p_1 and p_2 takes $O(n)$ time.



- To find each next hull vertex p_i, we spend $O(n)$ time.

Time Complexity of Jarvis's March



- If the number of extreme points(and boundary edges) is k, then the time complexity of Jarvis's march is $O(nk)$.

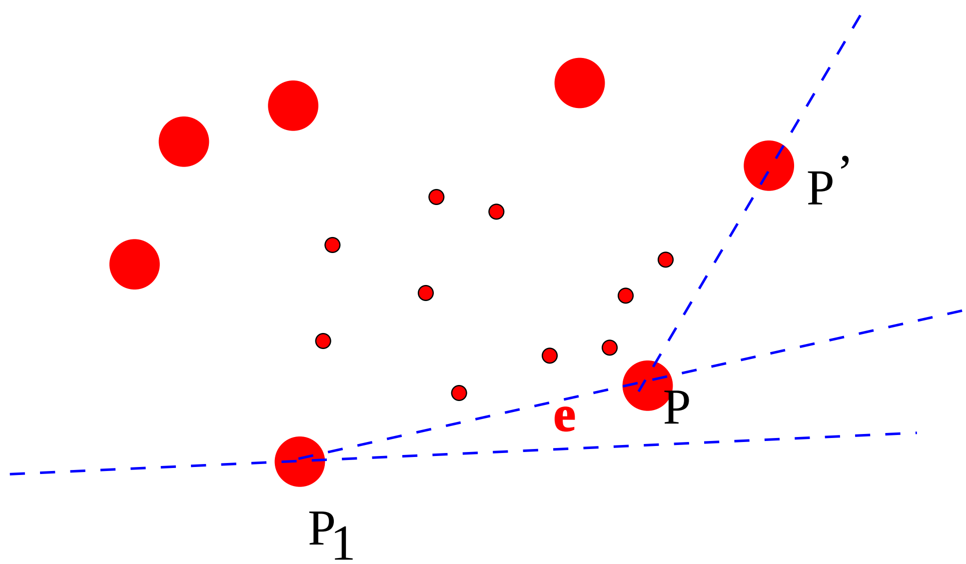
- If the number of extreme points k is small compared with $O(n)$, i.e., if k is bounded by a constant, then Jarvis's March runs in linear time.

- Jarvis's march can be generalized to higher dimensions.

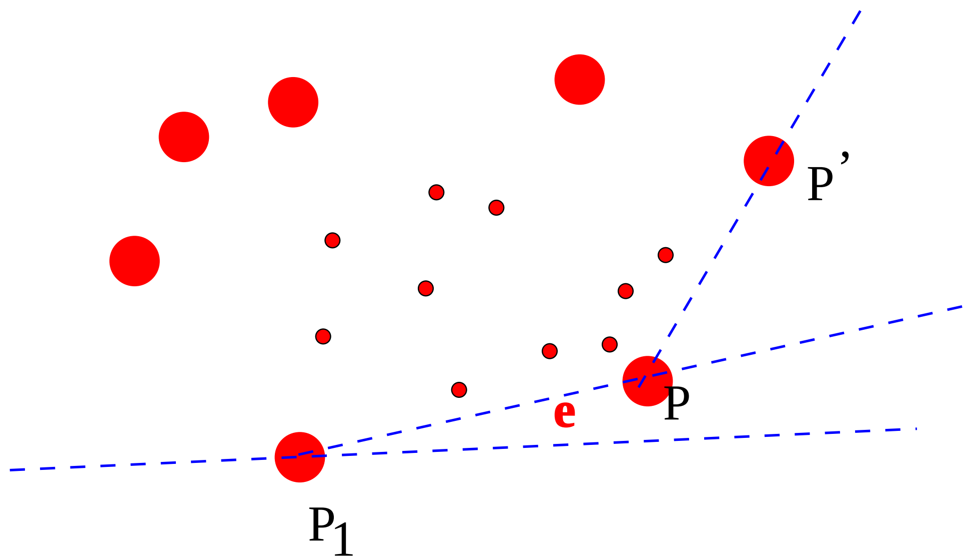
How to improve Jarvis's Algorithm

In Jarvis's algorithm, each time

- Based on the recent hull vertex p and the most recent hull edge e, we find the next hull vertex by choosing the point p' which makes the angle between e and pp' largest.



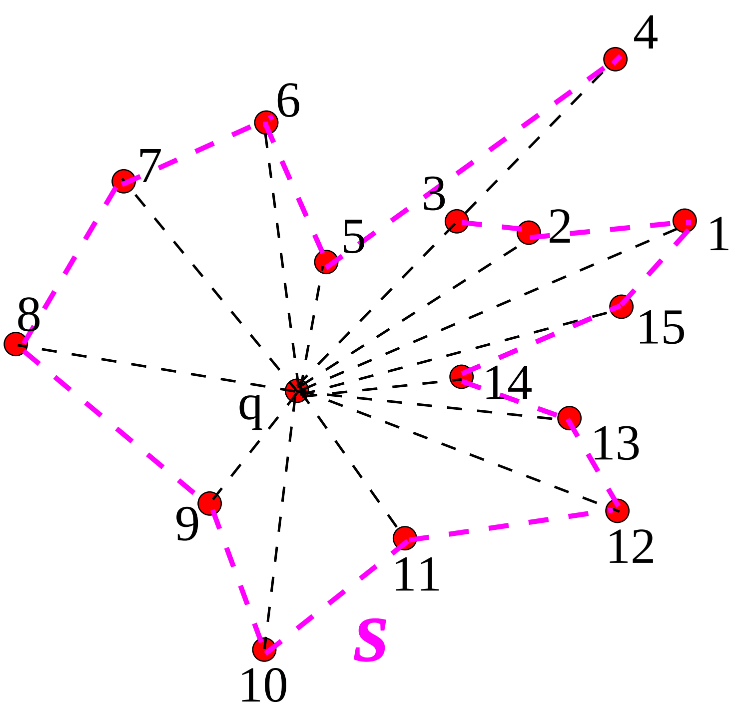
How to improve Jarvis's Algorithm



- A possible improvment is that we presort the points in some way so that once we find a point is not qualified for the next hull vertex, then we exclude the point forever.

Algorithm-Graham's Scan (1972)

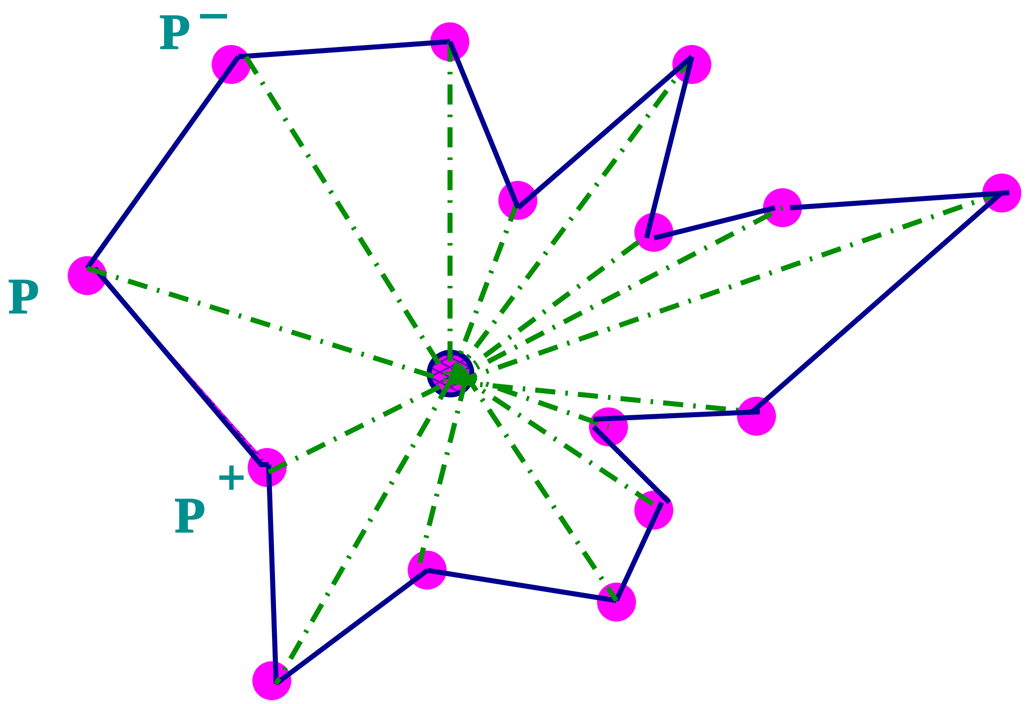
- Determine an interior point q of $CH(P)$.
- Sort the points of P around q by non-decreasing polar angles. If several points of P have the same polar angle, sort by increasing distance from q.



- Let S be the polygon through all the points of P so that the appear in the sorted order in the counterclockwise traversal of P.

Graham's Scan

- Identify the leftmost point of P. Denote it by p(p^- be the predecessor of p and p^+ be the successor of p on P).

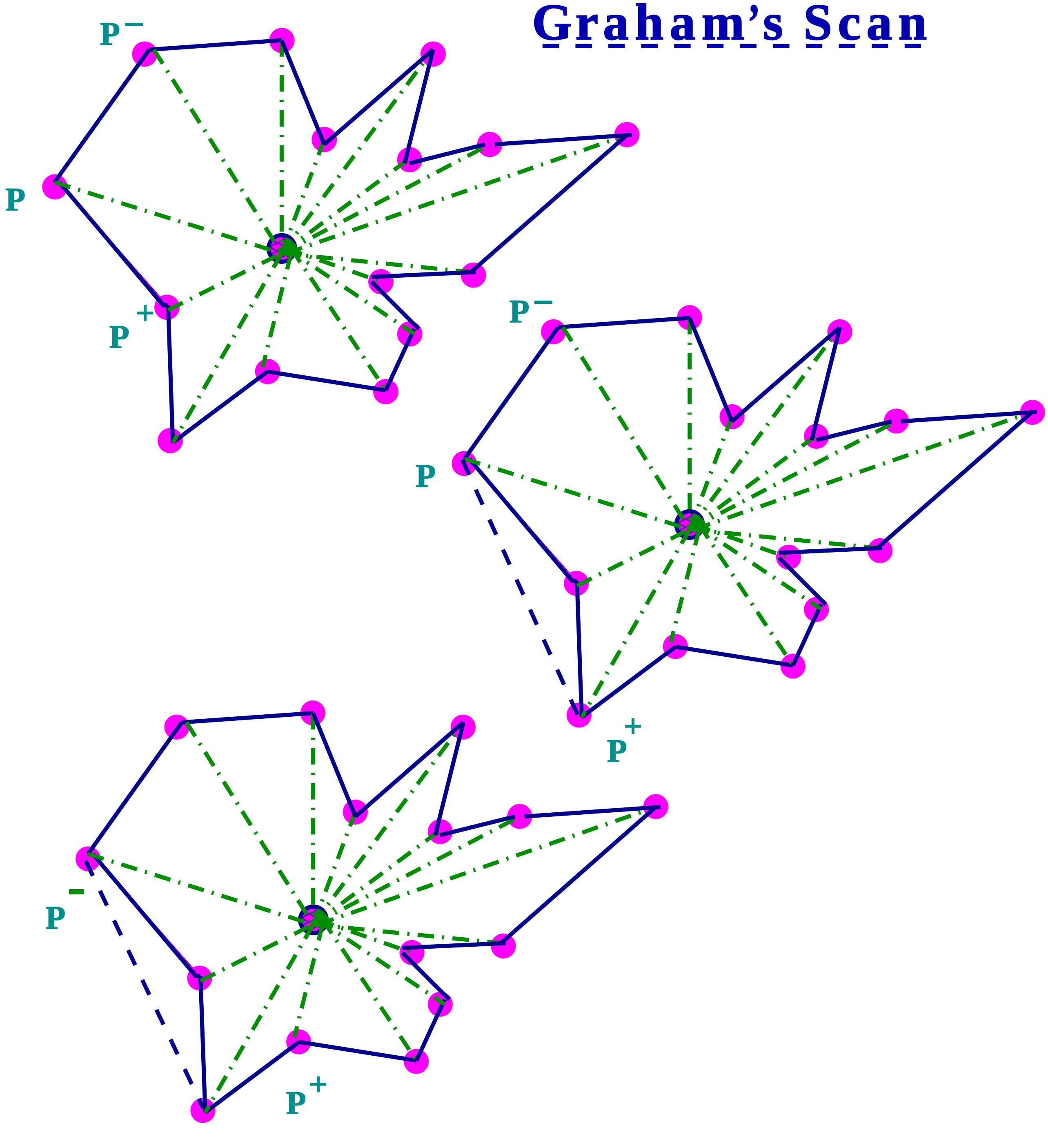


If $\{p^-, p, p^+\}$ **makes a left turn at** p

then $p := p^+$

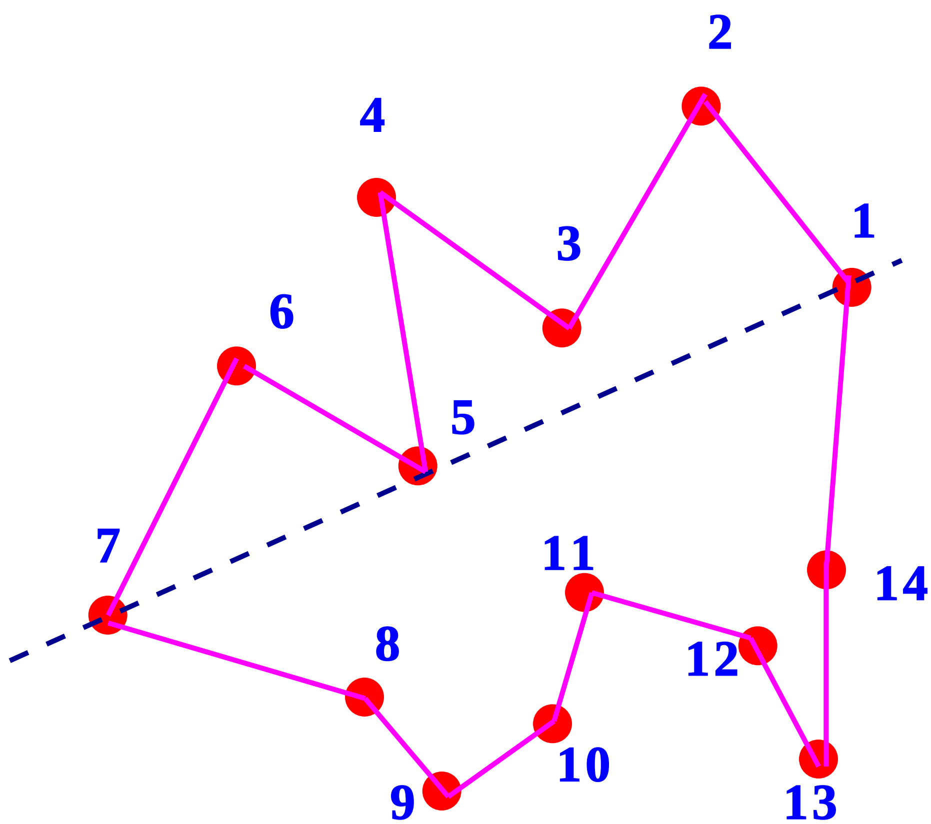
else remove p **and set** $p = \bar{p}$

Graham's Scan



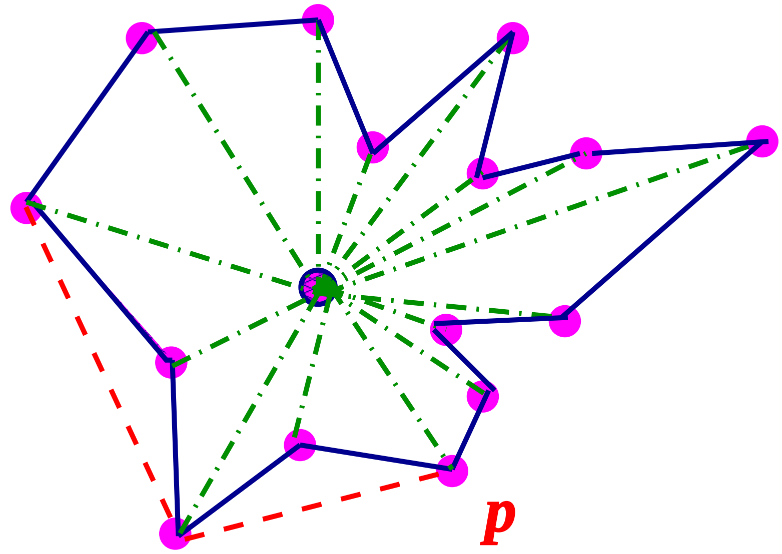
Graham's Scan-Continued

- It is not necessary to determine q.



Graham's Scan-Correctness and Complexity

- Graham's scan will never go backward behind the initial leftmost point of P.
- When arriving at some point p, all points between the initial point and p are left turns on the polygon.



Graham's Scan-Correctness and Complexity-Continued

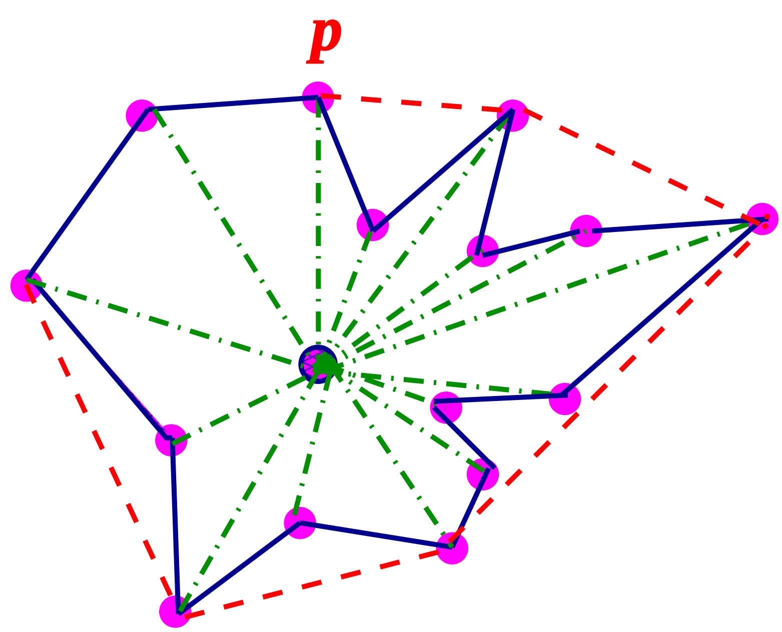
- After arriving the initial leftmost point (by forward step), P has left turns only (it is convex).

- Number of backward steps is $O(n)$: during each backward step one point is removed from P.

- Number of forward steps is also $O(n)$: Since there are only $O(n)$ points in the set.

- Both forward and backward steps require $O(1)$ time.

Graham's Scan-Correctness and Complexity-Continued



- Graham's scan requires $O(n)$ time after the points of P have been sorted in $O(nlogn)$ time.

- Graham's scan can be regarded as a modification of point pruning.

Convex Hulls in the Plane - Summary

- **Point pruning ....** $O(n^4), O(n^2)$

- **Edge Pruning ....** $O(n^3)$

- **Jarvis's march ....** $O(nh)$

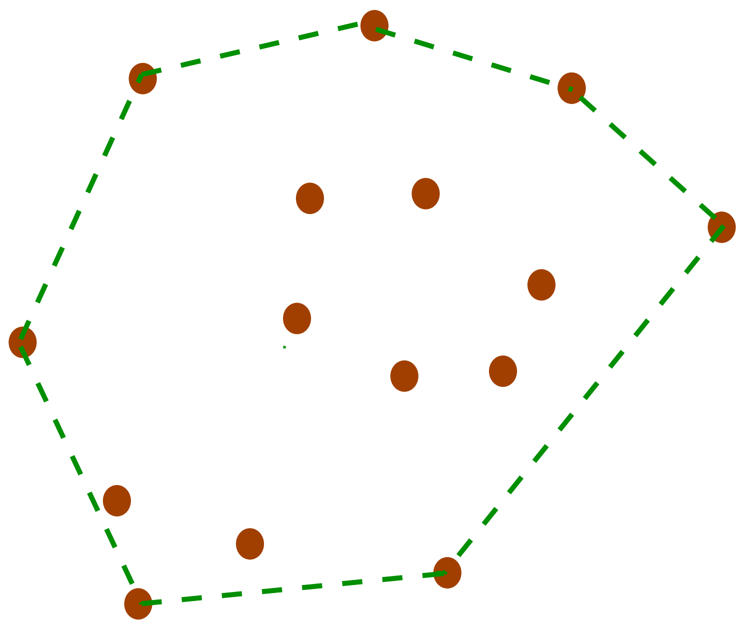
- **Graham's scan ....** $\Theta(nlogn)$

An overview of Lecture 8

- Review of Lecture 7
- Quick Hulls
- Divide and Conquer
- Randomized Incremental Sorting Algorithm
- Randomized Incremental Convex hull Algorithm
- Summary

Review of Lecture 7

CONVEX HULL : Given an arbitrary set P of n points of E^d, the *convex hull* $CH(P)$ of P is the smallest convex set containing P.



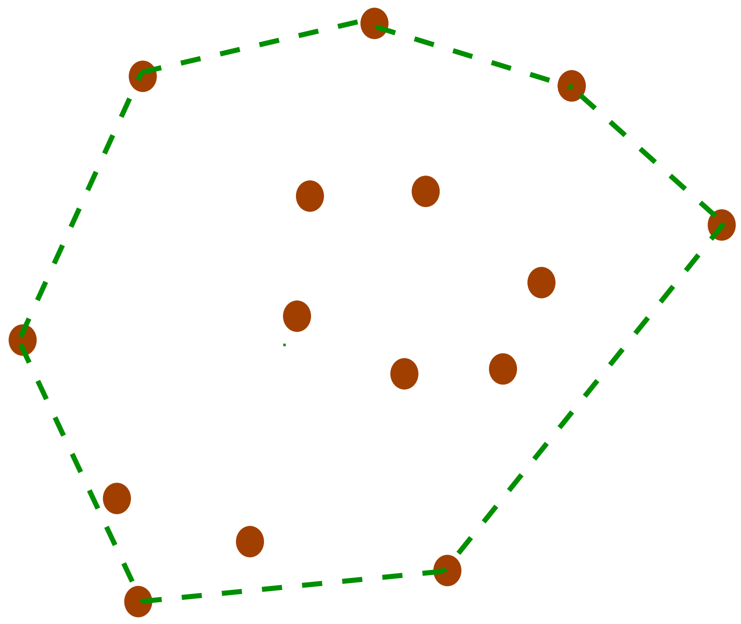
- The set E of extreme points is the smallest subset of P having the property that $CH(P) = CH(E)$ and E is precisely the vertices of P

Review of Lecture 7

Two steps are required to find the convex hull of a finite set:

(1). Identify the extreme points.

(2). Order these points so that they form a convex polygon.



Convex hulls again

If improvements are to made in the algorithm, them must come

- either by eliminating redundant computations (Point pruning, Edge pruning, Jarvis's march and Graham's scan).

- or by taking a different theoretical appraoch.

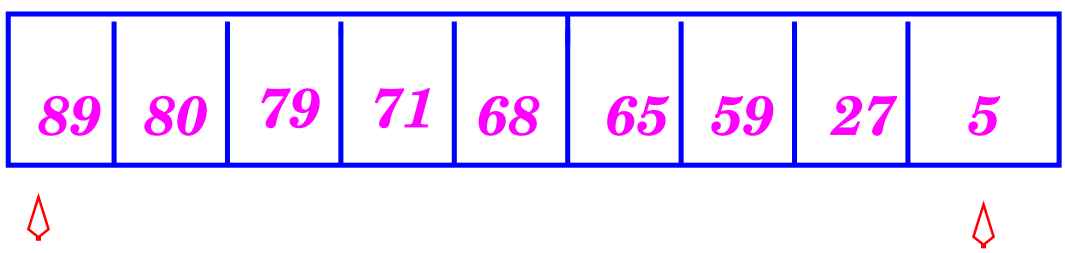
  Divide and Conquer algorithms :

  - *Quickhull*
  - *Mergehull*
  - *Randmozied Incremental Convex Hull*

Quickhull Techniques

Quicksort : Given an array of n numbers ,

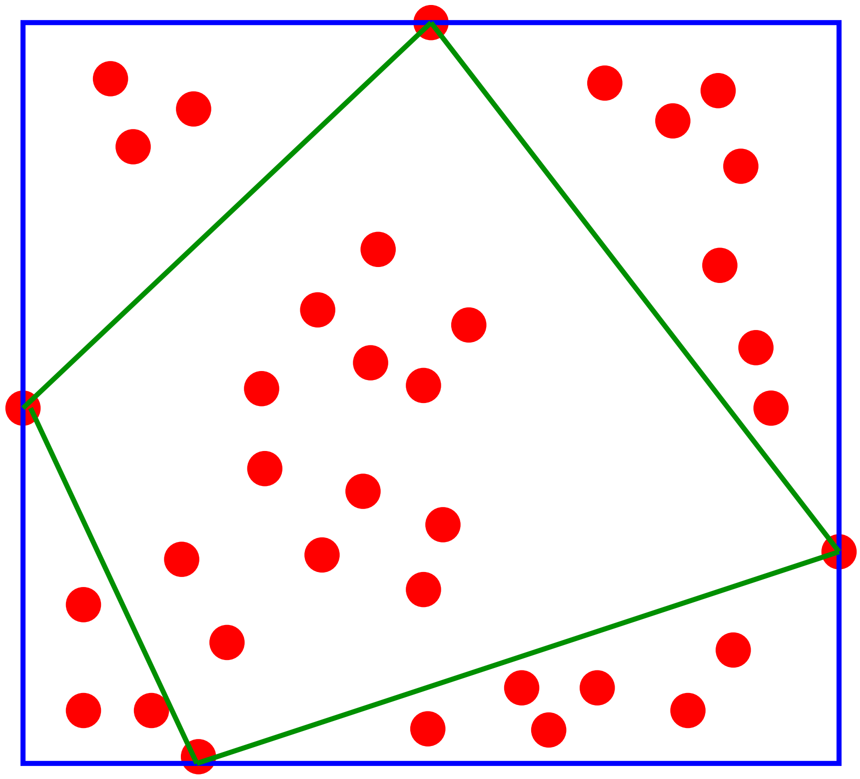
- partition it into a left and right subarry, such that each number in the first is less than each number of the second.
- recursively call the above subroutine.
- merge the two sorted sublists.



- Quickhull is the analogue of the Quicksort algorithm.

Quickhull Algorithm(1977)

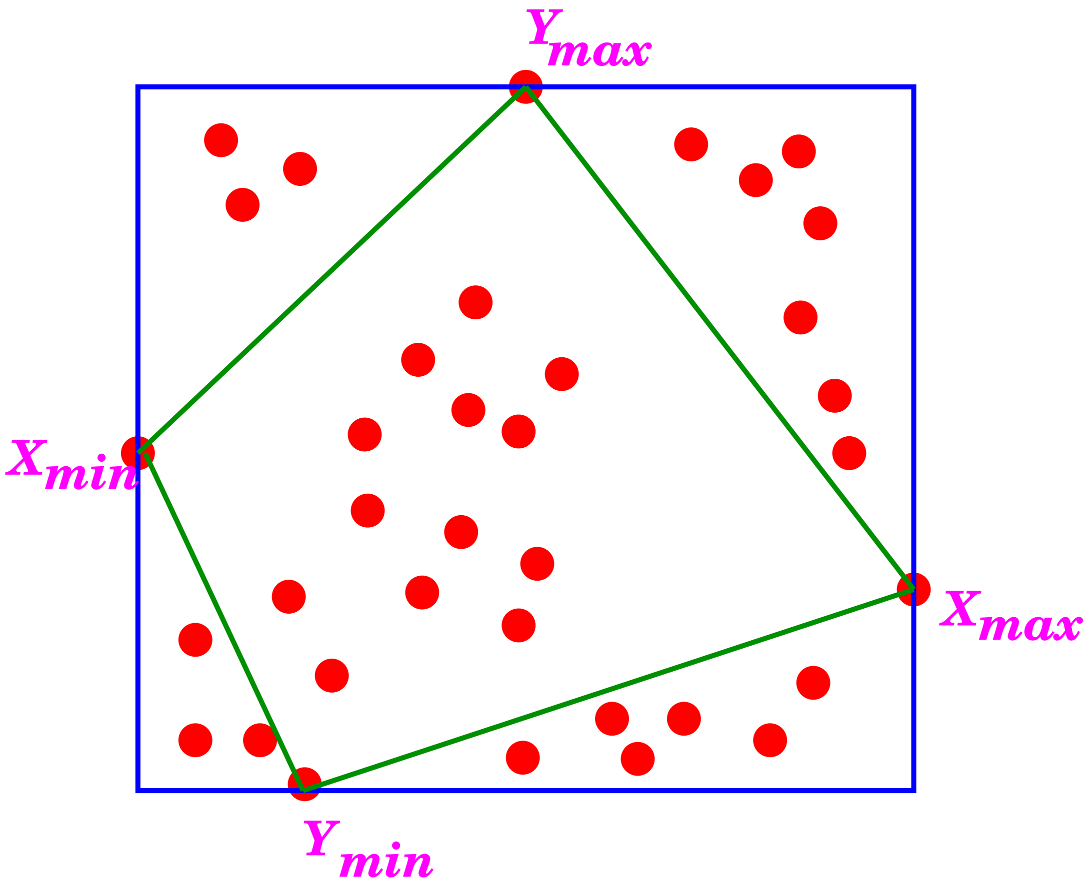
- General idea: discard the points that are not on the convex hull as quickly as possible.



QuickHull's Initial quadrilateral

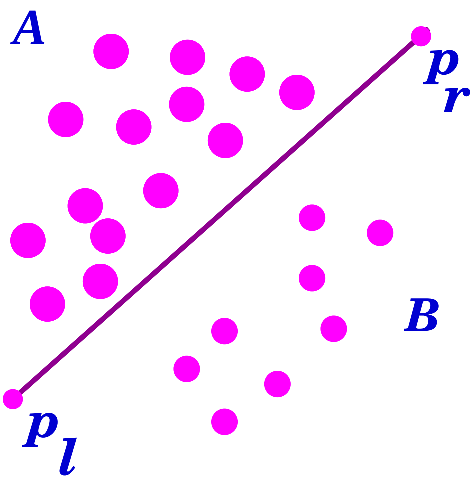
Quickhull Algorithm

- First compute the points with maxmimum and minimum x- and y-coordinates.
- The points lying within the quadrilateral $X_{min}Y_{min}X_{max}Y_{max}$ can be elimated in $O(n)$ time.
- Classify the remaining points into four corner triangles.



Quickhull Algorithm - Continued

QuickHull Elimination Procedure:

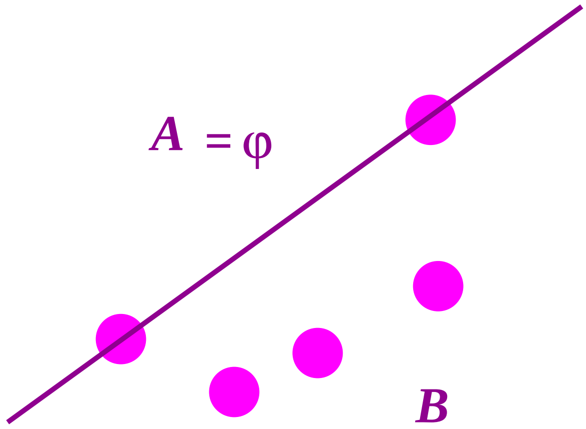


- Find the leftmost and the rightmost points p_l and p_r.

- Partition the remaining points into two subsets A and B depending on whether they are above or below the line L through p_l and p_r.

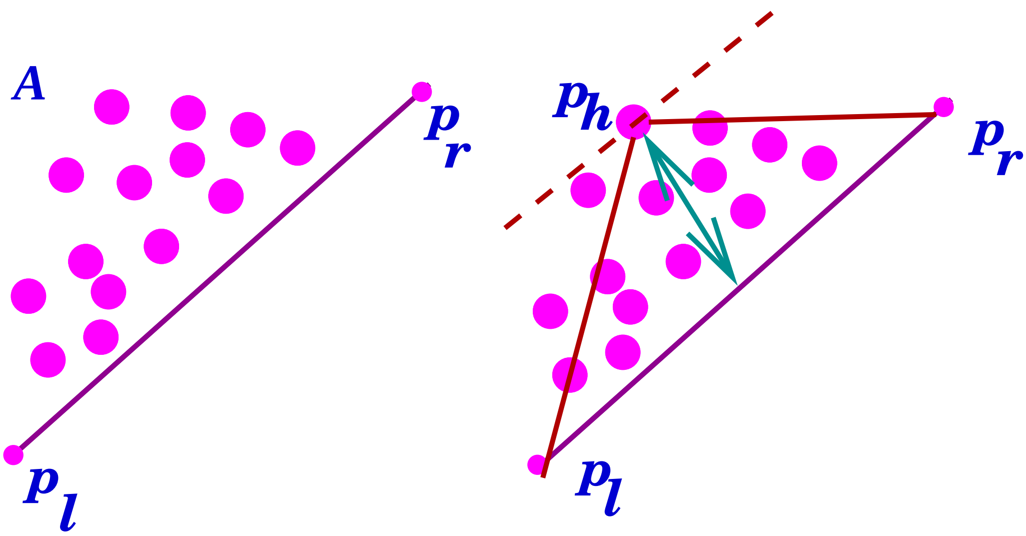
UpperHull(A, p_l, p_r)

- Consider A. If $A = \emptyset$, then $p_l p_r$ is a boundary edge.



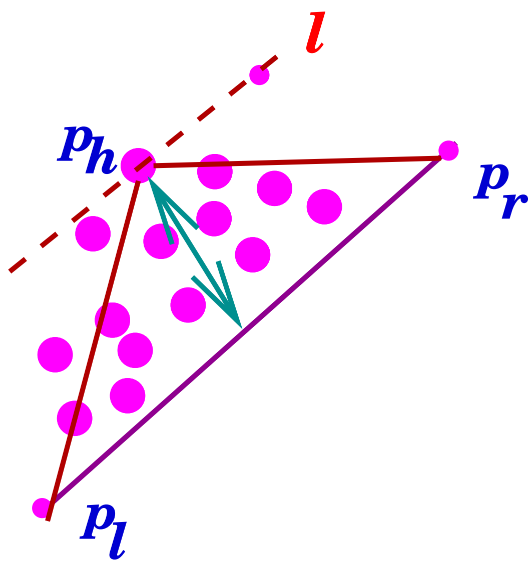
UpperHull(A, p_l, p_r)

- If $A \neq \emptyset$, then determine a point p_h such that $\Delta p_l p_r p_h$ is largest possible. If there are several candidates for p_h, select the leftmost one.



UpperHull(A, p_l, p_r)

p_h is an extreme point of $CH(P)$.

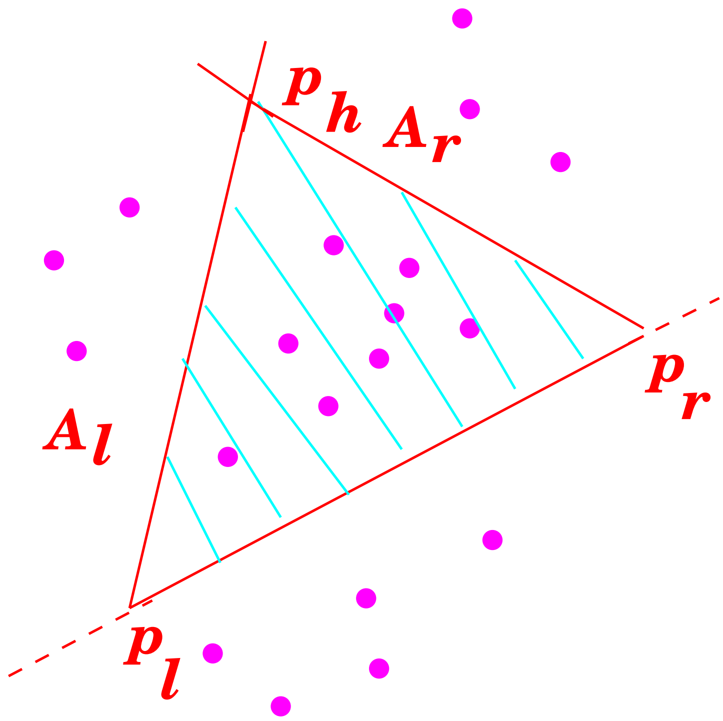


QuickHull

- Prune the points of P in $\Delta p_l p_r p_h$.

- Subdivide the remaining points in A into two subsets A_L and A_R by drawing the lines through p_l and p_h as well as through p_r and p_h and repeat for A_L and A_R.

- Repeat for B.



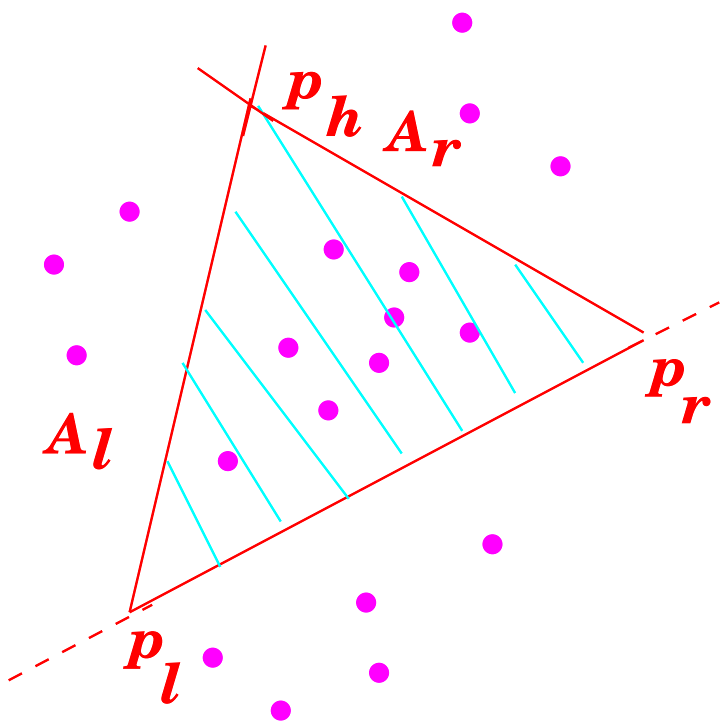
QuickHull-Algorithm

UpperHull(P, p_l, p_r)

begin
If $P = \{p_l, p_r\}$ then return (p_l, p_r) .
else **begin**
p_h := Furthest(P, l, r) ;
(furthest to line $p_l p_r$)
A_L := points of P on or to the
left of $(\vec{p_l, p_h})$
A_R := points of P on or to the
right of $(\vec{p_h, p_r})$
(Recursively call UpperHull(A_L, p_l, p_h) and
UpperHull(A_R, p_h, p_r))
return UpperHull(A_L, p_l, p_h) *
UpperHull(A_R, p_h, p_r)
end
end.

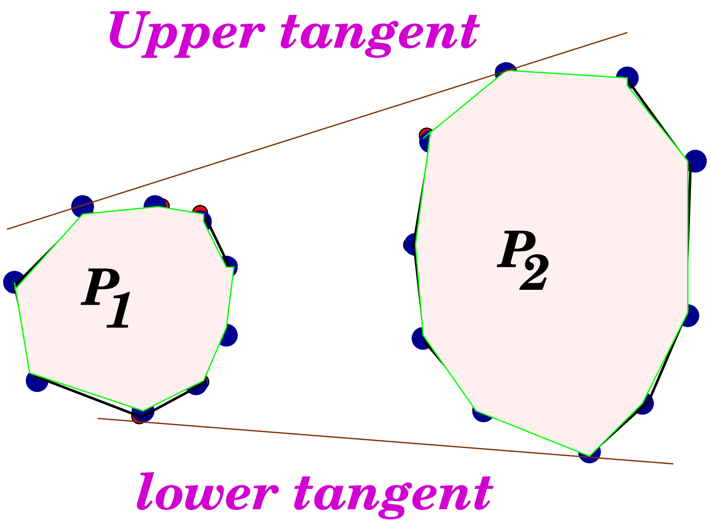
QuickHull-Complexity

- The extraction from P of A (and B) including the elimination of points internal to the triangle $\Delta p_l p_r p_h$ carried out in $O(n)$ time.
- If the size of A and B is at most equal and this holds at each level of recursion, complexity $O(nlogn)$.
- Worst case complexity : $O(n^2)$ as partitioning can be very uneven.

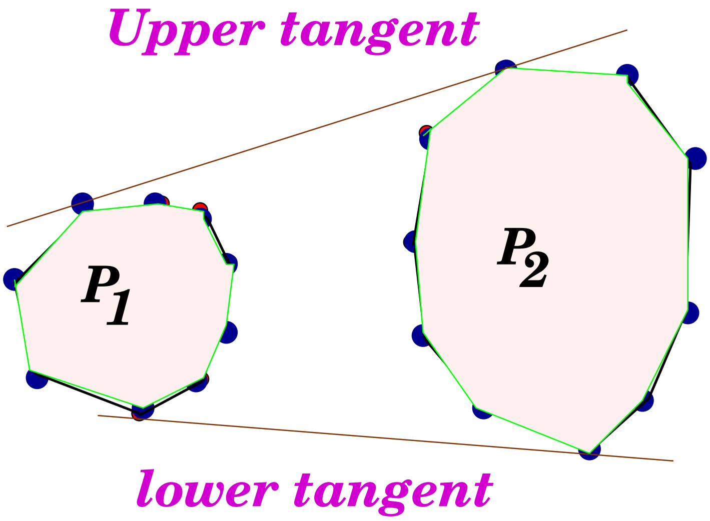


Convex Hull by Divide and Conquer

- $O(nlogn)$ algorithm.
- can be viewed as a generalization of merge sort



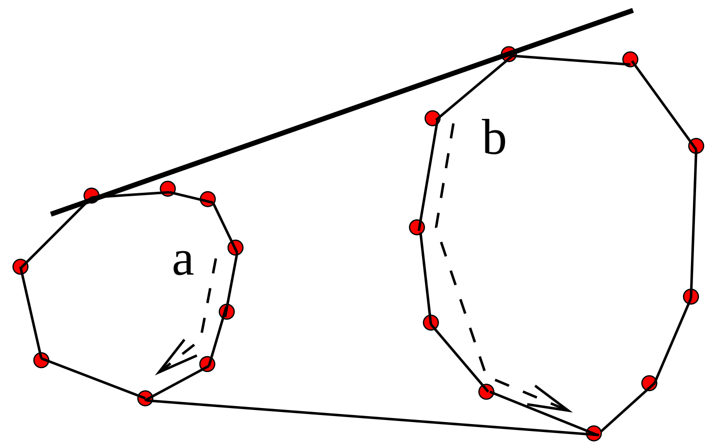
Divide and Conquer(1978)



- Solve directly if $|P| \leq 2$. **Return** .

- Partition P into two " equal size " subsets P_1 and P_2. where P_1 consists of points with the lowest x-coordinates and P_2 consists of the points with the highest x-coordinates.

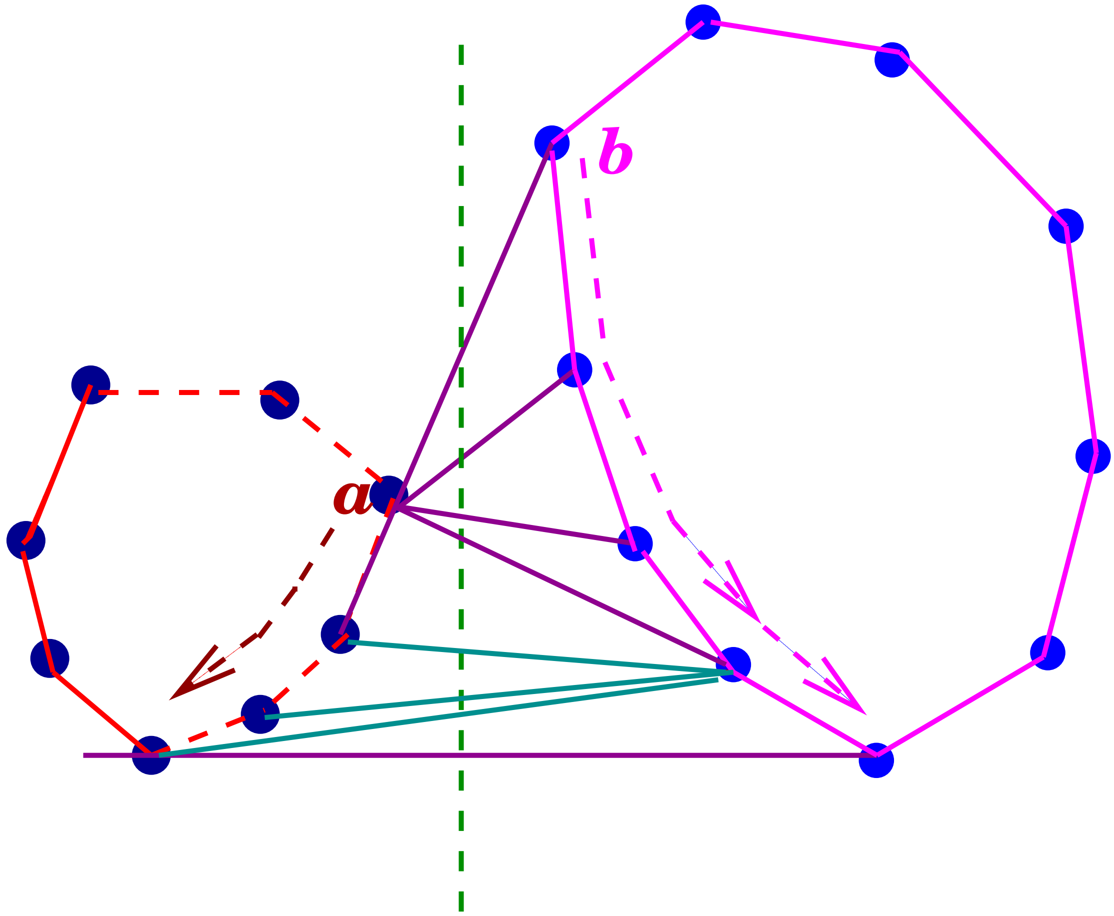
Divide and Conquer-Continued)



- Determine (recursively) $CH(P_1) = H_{P_1}$ and $CH(P_2) = H_{P_2}$.

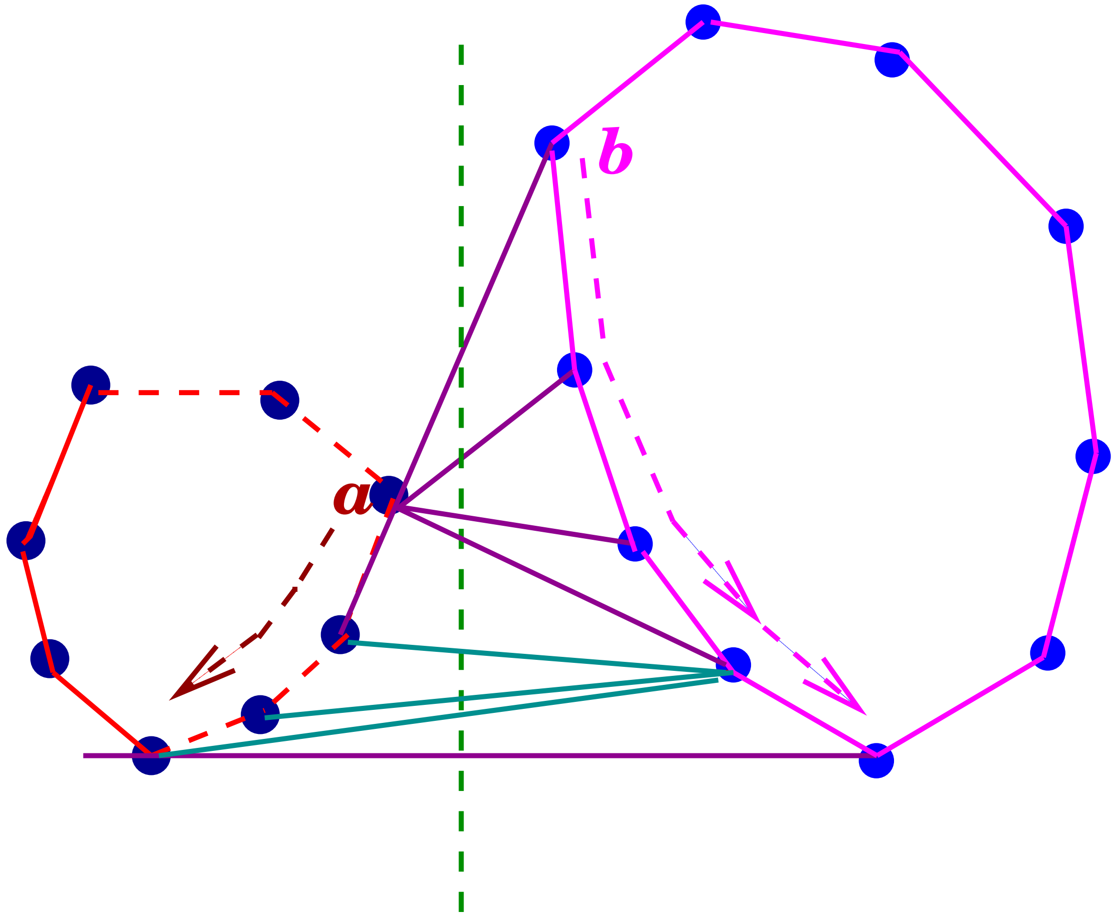
- Merge the two solutions to obtain $CH(P)$, by computing the upper and lower tangents for H_{P_1} and H_{P_2} and discarding all the points lying between these two tangents.

Computation of Lower Tangent



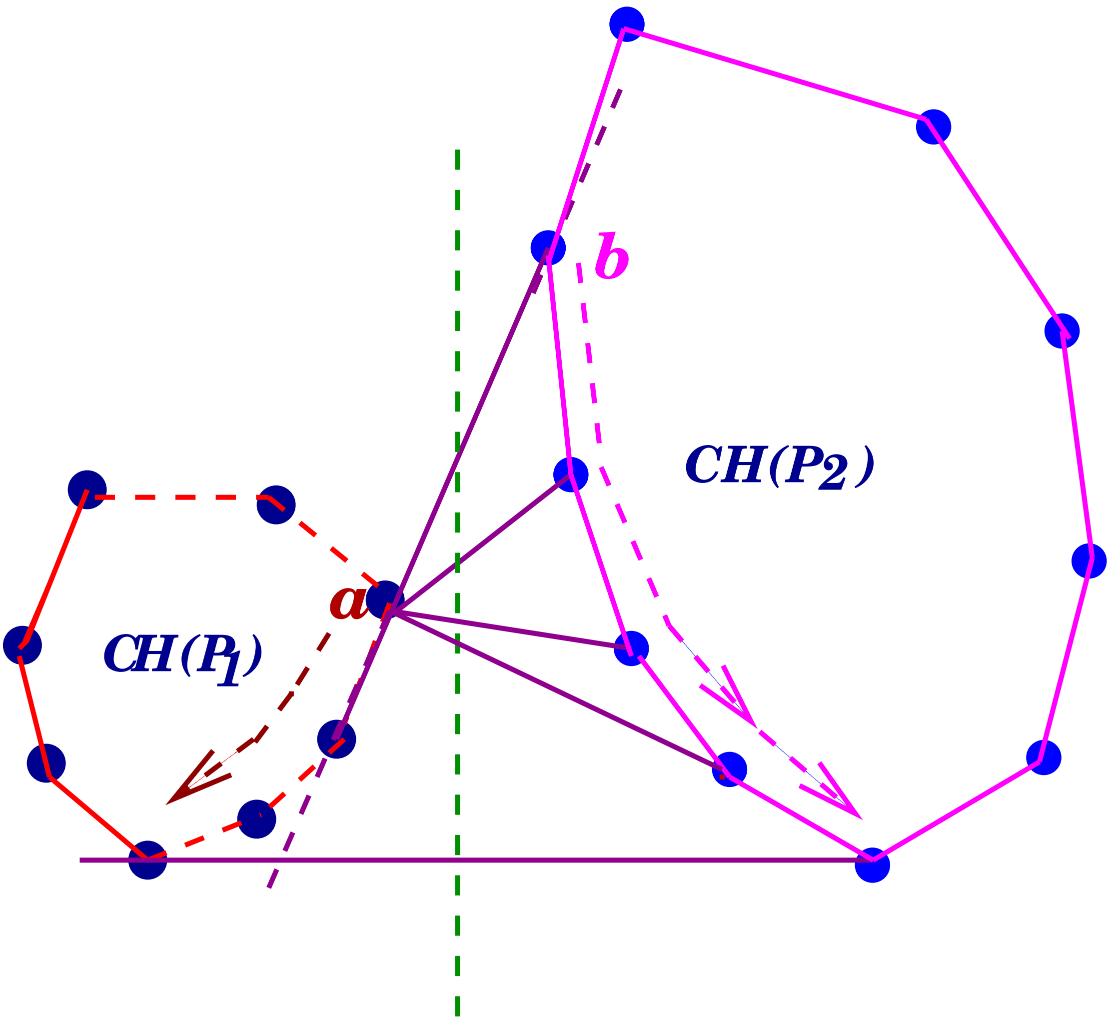
- Initialize a to be the rightmost point of P_1 and b is the leftmost point of P_2 (The points a and b can be found in $O(n)$ time).

Computation of Lower Tangent



- Lower tangency is a condition that be tested locally by an orientation test of the two vertices and neighboring vertices on the hull.

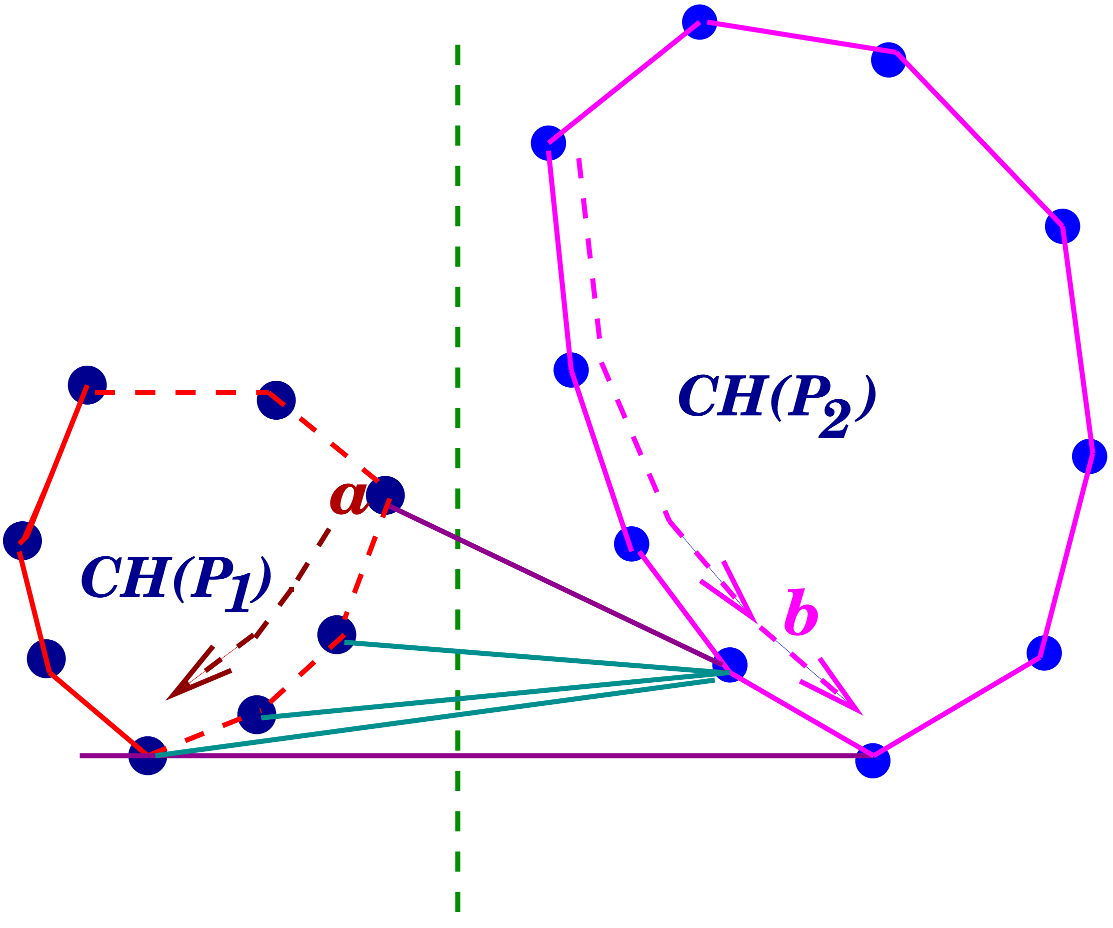
Computation of Lower Tangent



***a b* is not a tangent to**

CH(P2)

Computation of Lower Tangent



***ab* is not a tangent to $CH(P_1)$**

Computation of Lower Tangent

Lower-Tangent(H_{P_1}, H_{P_2});
(1) a:=rightmost vertex of H_{P_1};
(2) b:=leftmost vertex of H_{P_2};
(3) **while** ab is not lower tangent of both H_{P_1} and H_{P_2}
do
 (a) **while** ab is not a lower tangent to H_{P_1}
 do $a := a - 1$;
 (move a clockwise)
 (b) **while** ab is not a lower tangent to H_{P_2}
 do $b := b + 1$;
 (move b counterclockwise)
Return ab.

The important thing is each vertex on each hull can be visited atmost once by the search, and hence the running time is $O(m)$ where $m = |H_{P_1}| + |H_{P_2}| \leq |P_1| + |P_1|$.

Time Complexity of Divide and Conquer

- Complexity:

$$f(n) = \begin{cases} O(1) & n = 2 \\ 2f(n/2) + O(n) & n > 2 \end{cases}$$

- It is well-known that such a recursive function is $n \log n$

- The tangents can be computed in $O(n)$ time.

Randomized incremental construction

- We use a technique called randomized incremental construction for designing a randomized algorithm for convex hull.

- This technique is very useful for designing randomized geometric algorithms.

- We use a random permutation of the input and the the resulting algorithm is a Las Vegas algorithm. It always produces the correct result.

- We will try to estimate the expected running time of the algorithm.

A randomized sorting algorithm

Input : A set of n unsorted numbers.

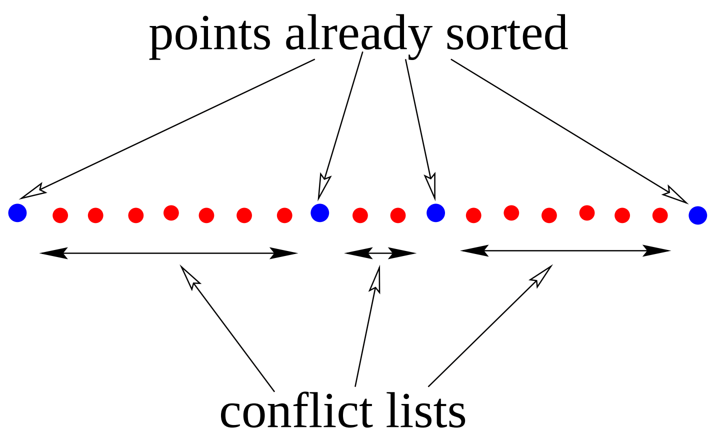
Output : A sorted set of these n numbers.

- We sort the numbers incrementally. At every step, a random input is chosen and added to the sorted set.

- Hence after step i, we have a sorted set of i numbers and an unsorted set of $n - i$ numbers.

- For adding the next input efficiently, we use the idea of a conflict list.

A randomized sorting algorithm



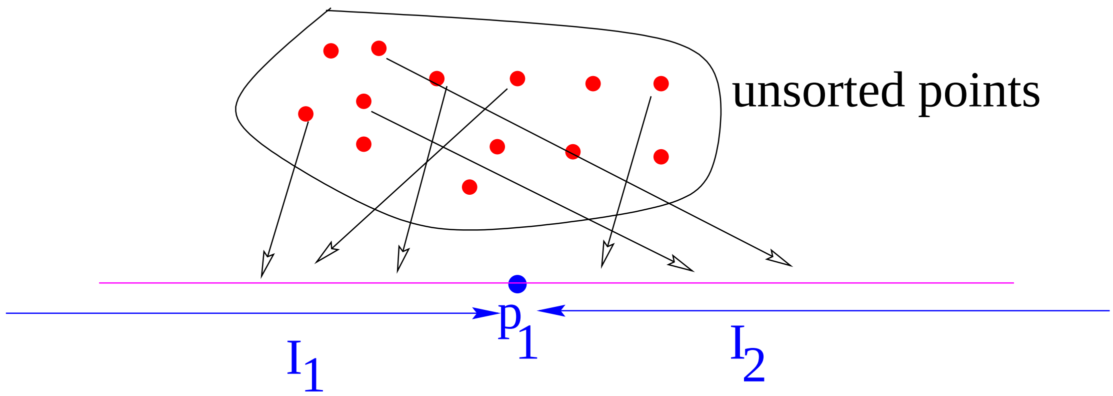
- After the i-th step, cosider the $n-i$ unsorted points.

- Each of these unsorted points will be in one of the $i+1$ intervals defined by the i sorted points.

- With each interval between two adjacent sorted points, we keep a list of all the unsorted points in that interval. This is called a conflict list.

How do we maintain the conflict lists?

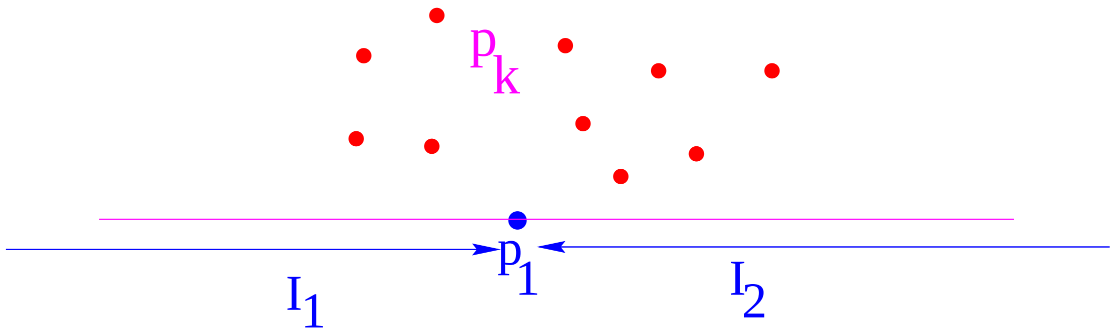
- Consider the first point p_1 that we choose from the n unsorted points.
- p_1 introduces two intervals I_1 and I_2 for all the unsorted points.
- We compare each unsorted point p_k with p_1 and keep a pointer either to I_1 or to I_2.



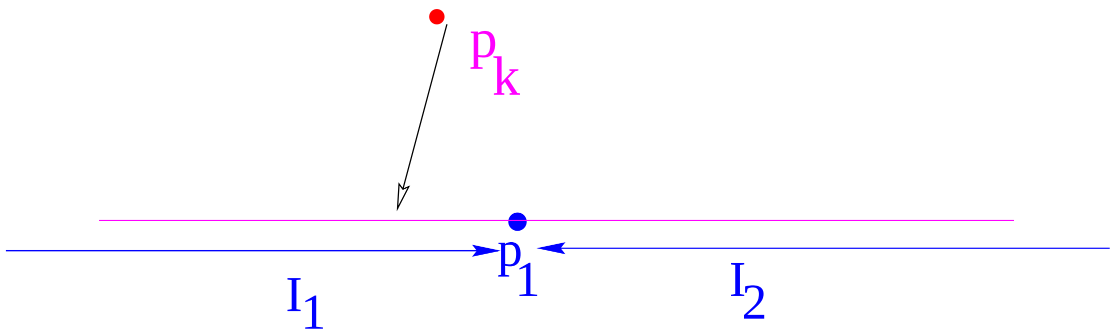
How do we maintain the conflict lists?

- We also keep a list of all the points that are in I_1 and in I_2. These are the conflict lists.

- Suppose we randomly choose p_k as the next point to be added to the sorted list.



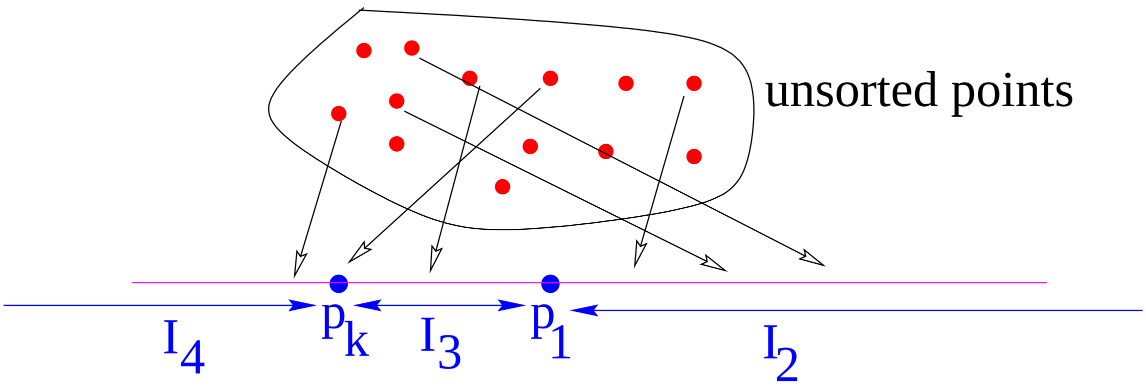
How do we maintain the conflict lists?



- From the pointer stored with p_k we can determine in $O(1)$ time, p_k should be added to which interval I_1 or I_2.

- Suppose p_k goes to I_1. I_1 is divided into two intervals I_3 and I_4 due to p_k.

How do we maintain the conflict lists?



- We have to create two new conflict lists for I_3 and I_4 from the conflict list for I_1.

- We do not need to do anything with the conflict list of I_2.

Maintaining Conflict List

- We maintain a pointer for each number yet to be inserted in the sorted list.

- After the i-th step, the pointer for each uninserted number specifies which of the $i+1$ intervals in the sorted list it would be inserted into, if it were next to be inserted.

- The pointers are bidirectional, so that given an interval we can determine the numbers whose pointers point to it.

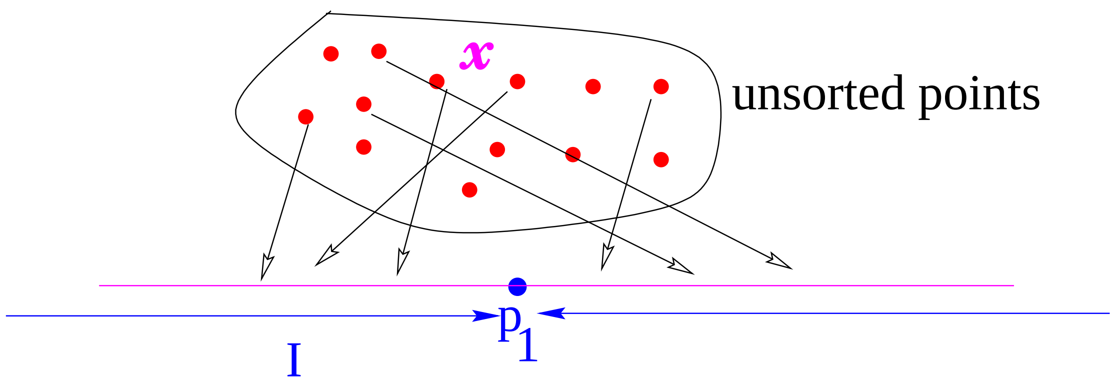
Updating the Conflict List

- what is the work required to maintain these pointers.
- Suppose we insert a number x whose pointer points to interval I.
- On inserting x, we have three tasks.

(1). find all numbers whose pointers point to I.

(2). update the pointers of all numbers whose pointers point to I.

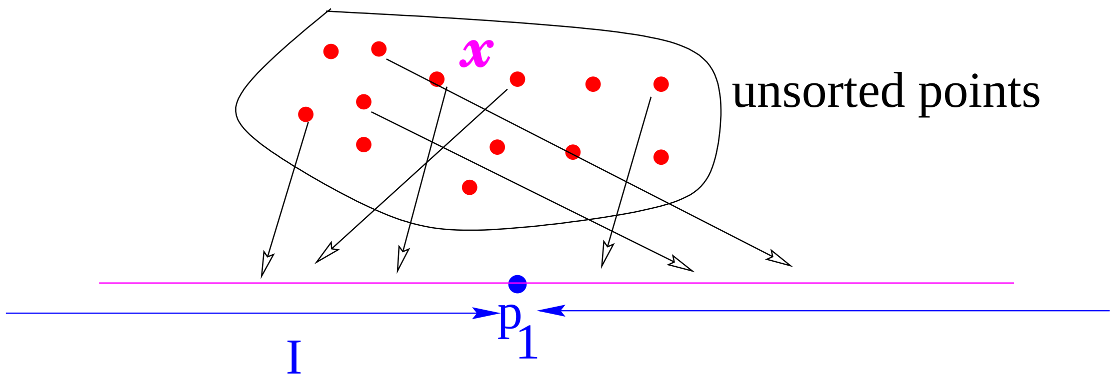
(3). delete the pointer from x to I.



Complexity for updating the conflict list

The important task is (2)..

- The work done in this update step is proportional to the number of pointers pointing to I.



Complexity analysis

- When we have added all the n inputs, we have the sorted set.
- We add a new random point at each step in $O(1)$ time, but we do a lot of work for changing the conflict lists.
- Suppose we have already added i points and we are trying to add the $i+1$-th point.
- The $i+1$-th step consists of choosing of one the $n-i$ yet unsorted numbers uniformely at random, and inserting it into the sorted list.

Complexity analysis

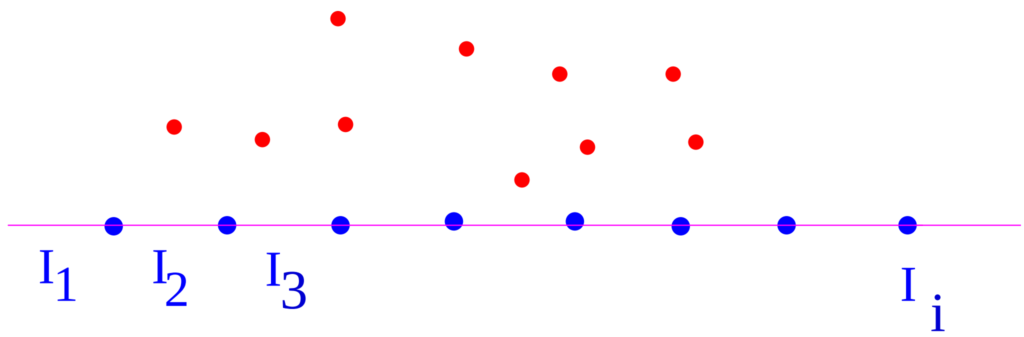
- We have to estimate what is the expected cost for the addition of the $i+1$-th point.
- We use a technique called backward analysis to estimate this.
- This has already been used in the course Design and Analysis of Algorithms(LP in two dimensions and constructing the trapezoidal decomposition for a set of line segments).

Backward analysis

- When we have a set of objects, it is easier to estimate the expected cost of choosing one object from the set.

- But it is difficult to estimate the cost of adding a new object which is not in the set.

Backward analysis



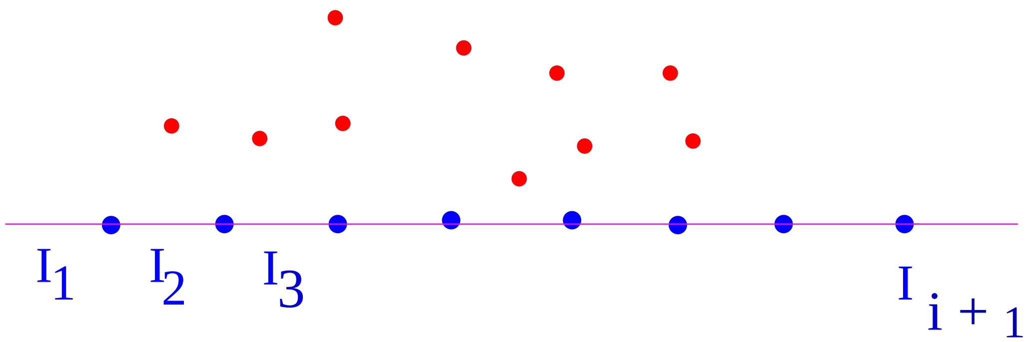
- In our case, if we want to estimate the cost of adding the $i+1$-th input to the sorted set of i points.

- But the $i+1$-th input is not in the sorted set of i points.

- So we go backwards!

Backward analysis

- We estimate the cost of deleting a random input from a sorted set of $i+1$ inputs.
- There are $n-i-1$ unsorted points and $i+2$ intervals before the deletion.

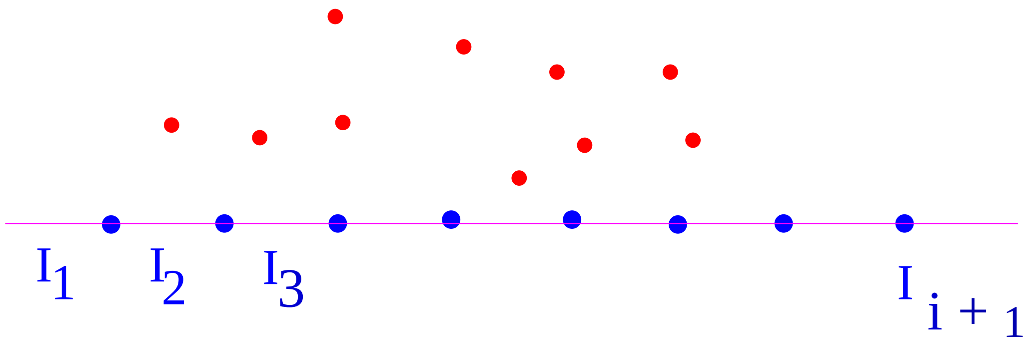


Backward analysis

- Remember that the numbers were added randomly in the original algorithm.

- So in the backward analysis we can assume that each of the $i+1$ numbers is equally likely to be deleted.

- After the deletion, there are $n-i$ unsorted points, i sorted points and $i+1$ intervals.



Backward analysis

- The expected number of unsorted points in one interval is : number of unsorted points divided by number of intervals. This is $\frac{n-i}{i+1} = O(\frac{n}{i})$.

- We have to change the pointers for all these points for updating the conflict list after the deletion of the $i+1$-th point.

- Hence this is the work done for the deletion of the $i+1$-th point.

Summing over all the steps, the expected total work is : $\sum_{i=1}^{n} O(\frac{n}{i})$

- From linearity of expectation, this is :

$$O(\sum_{i=1}^{n} n/i) = O(n \sum_{i=1}^{n} 1/i) = O(n \log n)$$

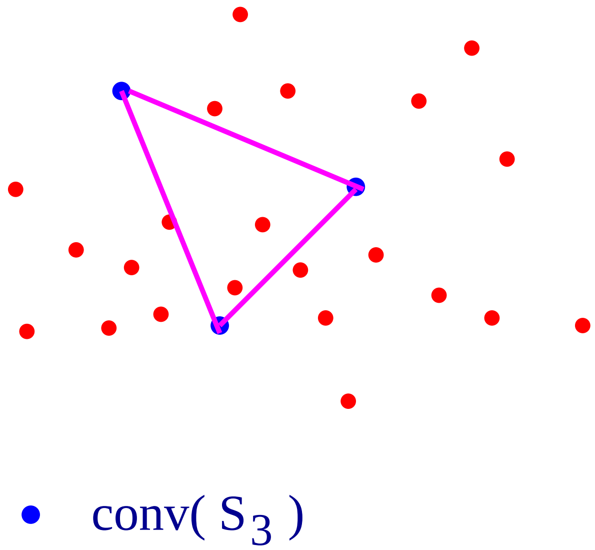
A randomized algorithm for convex hull

Input : A set $P = p_1, p_2, \ldots, p_n$ of n points.

Output : The convex hull of the n points.

begin

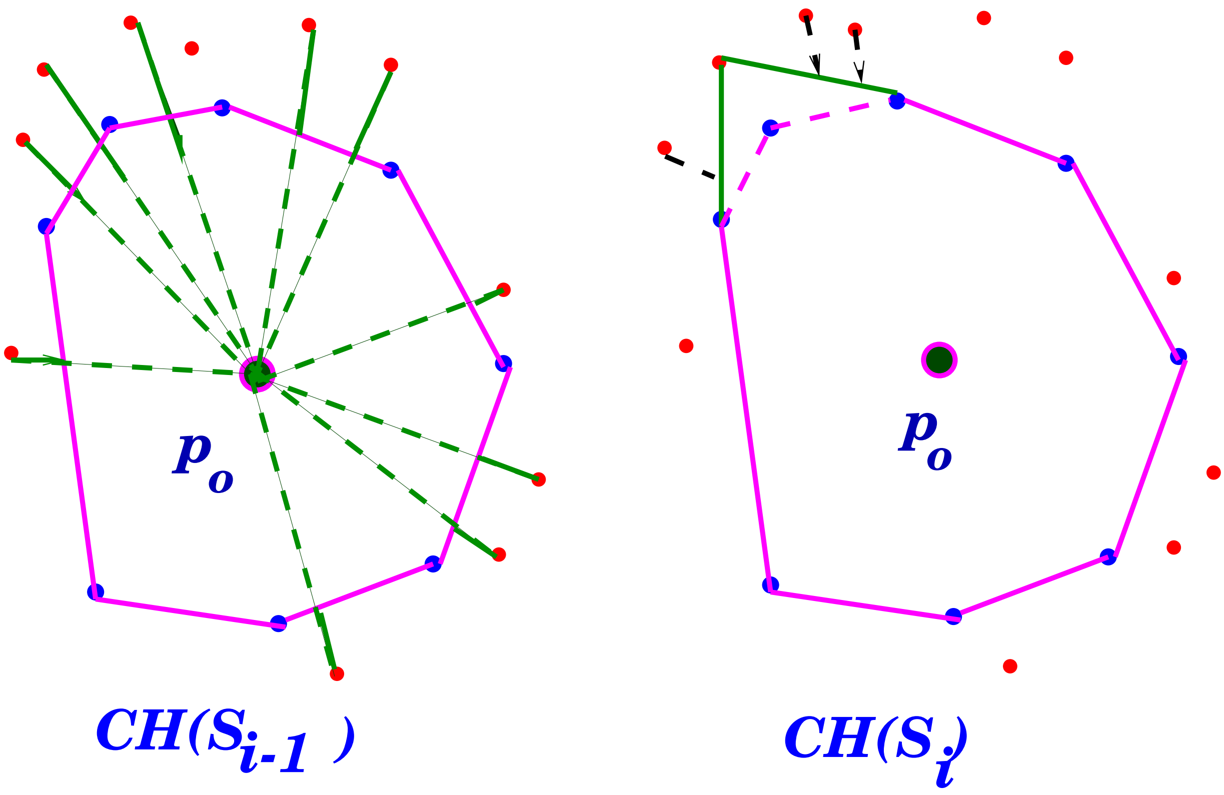
1. Choose any three points from the input and construct the covex hull $conv(S_3)$.



A randomized algorithm for convex hull

2. Do the following for $n-3$ time steps :

Add a randomly choosen point to the existing convex hull and update the convex hull.



Incremental convex hull

A randomized algorithm for convex hull

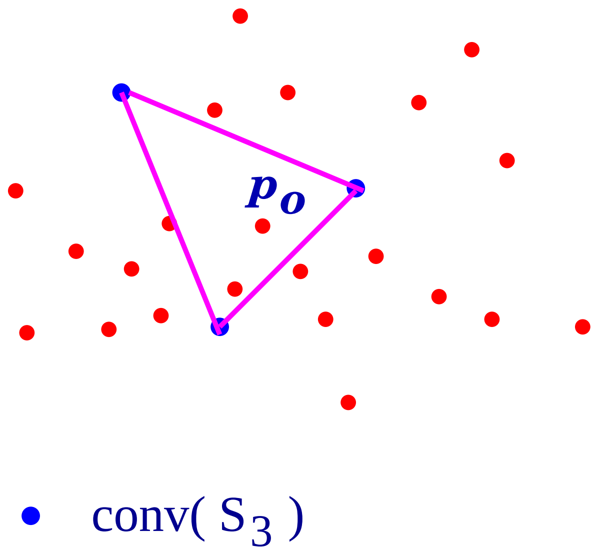
We have to specify :

- How do we add a randomly chosen point correctly?
- What are the conflict lists in this case?
- How do we update the conflict lists?

Initial Conflict List

For creating the conflict lists :

- We choose a point p_0 inside $conv(S_3)$.
- Connect the $n-3$ points to p_0.



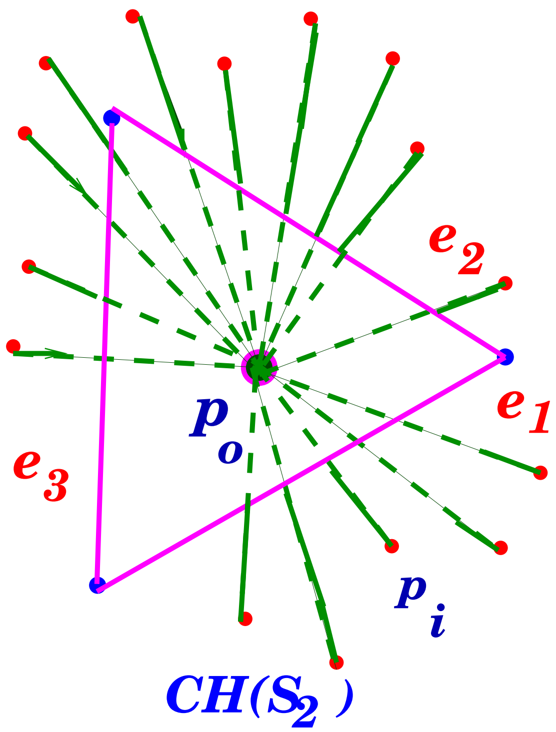
Conflict lists

- For a point p_i, if the line $\overline{p_0 p_i}$ intersects edge e_1,

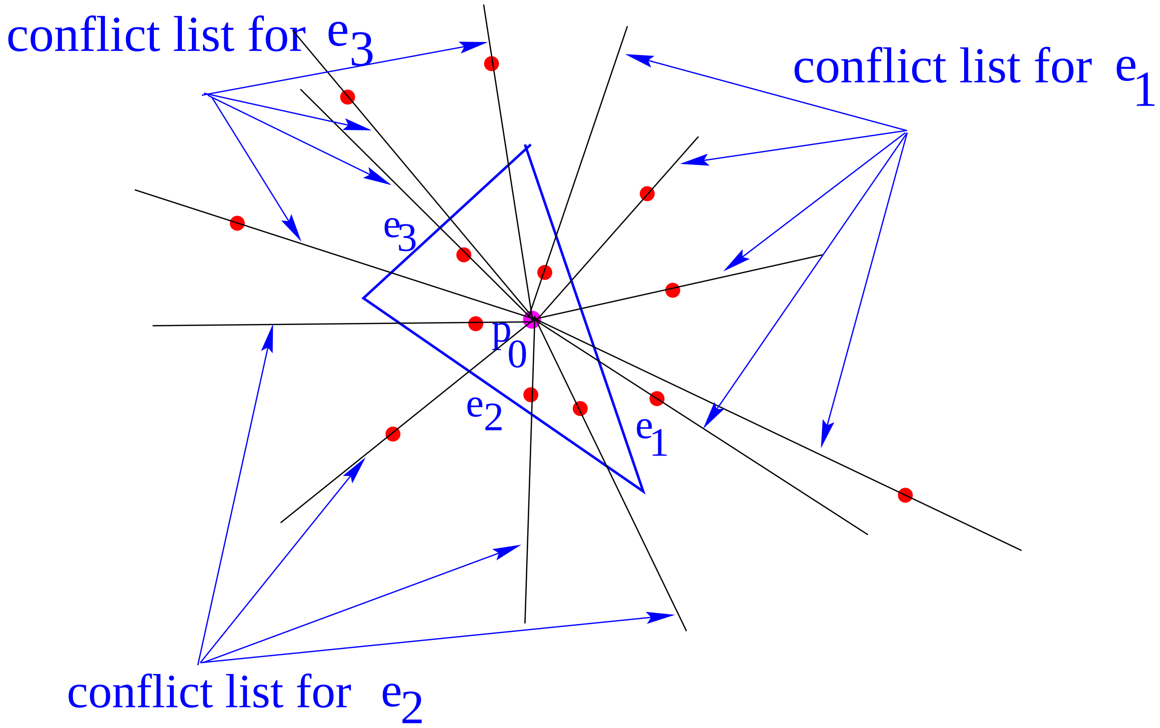
  (1). We keep a pointer to e_1 with the point p_i.

  (2). We include p_i in the conflict list of e_1.

- So we start with three conflict lists.

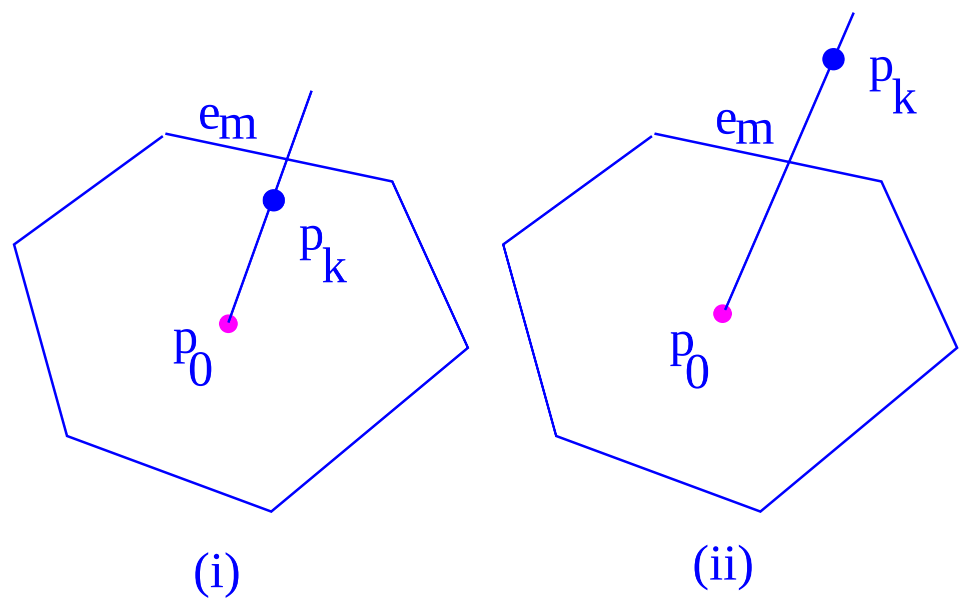


Conflict lists



Adding a new point

Suppose we are adding a new point p_k to $conv(S_i)$ which is the convex hull with i points.



- In $O(1)$ time we can determine that p_k belongs to the conflict list of edge e_m.

- In the first case, p_k is on the same side of e_m as p_0. Since p_k is inside $conv(S_i)$, we reject p_k.

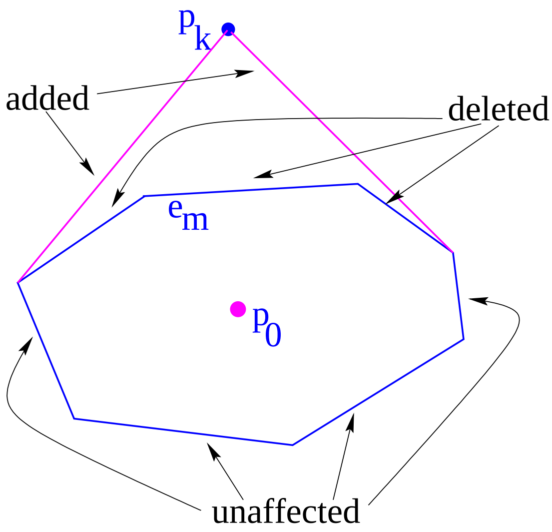
Adding a new point

- We are trying to construct $conv(S_{i+1})$ from $conv(S_i)$ after the addition of p_k.
- There are three kinds of edges after p_k is inserted :

(1). Edges which are unaffected

(2). Edges which should be deleted

(3). Edges which should be added



Adding a new point

- We keep the convex hull vertices in a doubly linked list so that we can move in both directions through the list.

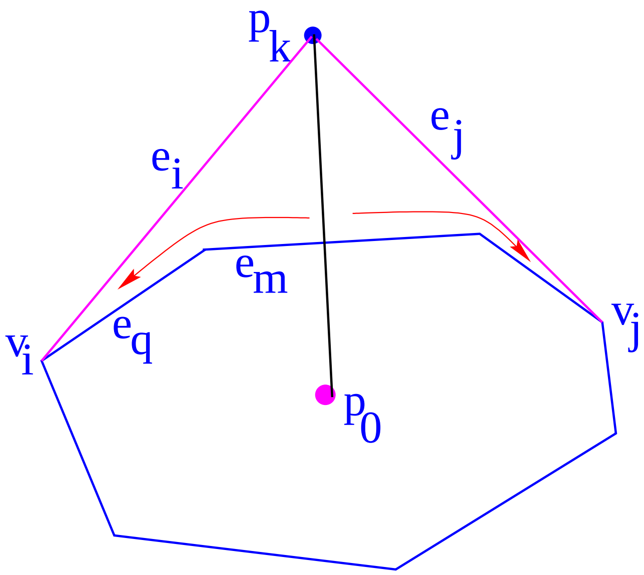
- Moving in both directions, we can find two vertices v_i and v_j such that $\overline{p_k v_i}$ and $\overline{p_k v_j}$ are tangents to $conv(S_i)$. Two new edges e_i and e_j are added.

- All the points in between v_i and v_j in $conv(S_i)$ are rejected.

- v_i and v_j are the neighbors of p_k in

  $conv(S_{i+1})$.

Adding a new point



- We have to update the conflict lists of all the edges we throw away.

- Consider an edge like e_q. All the points which are in the conflict list of e_q, should be included in the conflict list of e_i.

- Each such point will be added either to the conflict list of e_i or to the conflict list of e_j in $O(1)$ time.

Complexity analysis

- At most two edges are created at each step. Hence, the total work done for creating or deleting edges is $2n$. An edge may e created once and deleted once.

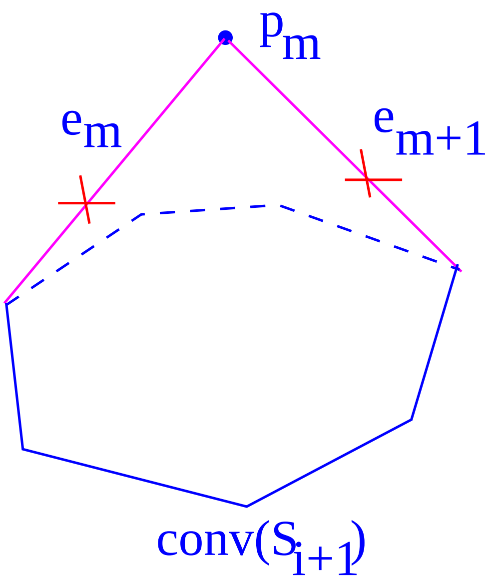
- The work done for adding a point p_k is proportional to

  (1). the work done for adding the point, and

  (2). the work done for updating the conflict lists

- We will estimate the expected work done through backward analysis.

Backward analysis



- In backward analysis, consider the deletion of a point from $conv(S_{i+1})$ to get $conv(S_i)$.
- Suppose we are deleting a point p_m. If we delete p_m, we have to delete two edges e_m and e_{m+1}.
- Since there are $i+1$ points in $conv(S_{i+1})$, the probability of choosing a point randomly is $\frac{1}{i}$.

Backward analysis

- There are $n - i$ points yet to be added to the convex hull.

- Hence, the expected number of points in the conflict list of edge e_m and e_{m+1} are $\frac{n-i}{i} = O(n/i)$.

- This is the expected work done for deleting the point p_m.

- Summing over all the steps, the expected total work is : $\sum_{i=1}^{n} O(\frac{n}{i})$

- From linearity of expectation, this is :

$$O(\sum_{i=1}^{n} n/i) = O(n \sum_{i=1}^{n} 1/i) = O(n \log n)$$

Convex Hulls in the Plane - Summary

- **Point pruning ....** $O(n^4), O(n^2)$
- **Edge Pruning ....** $O(n^3)$
- **Jarvis's march ....** $O(nh)$
- **Graham's scan ....** $\Theta(nlogn)$
- **Quickhull ....** $O(n^2)$

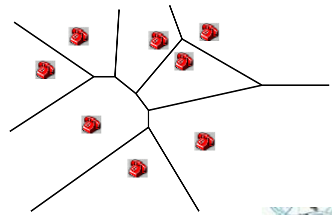
Convex Hulls in the Plane - Summary

- **Divide and Conquer ....** $\Theta(nlogn)$
- **Randomized Incremental ....** expected time complexity $O(nlogn)$

Voronoi Diagrams

- Definition
- Characteristics
- Size and Storage
- Construction
- Use

The Voronoi Diagram



Voronoi Regions

Eucledian distance :

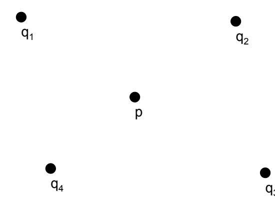
$dist(p,q) := \sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of n distinct points in a plane. We define the voronoi diagram of P as the subdivision of the plane into n cells, with the property that a point q lies in the cell corresponding to a site p_i iff $dist(q,p_i) < dist(q,p_j)$ for each $p_j \in P$ with $j \neq i$.

We denote the Voronoi diagram of P by $Vor(P)$.
The cell that corresponds to a site p_i is denotd by $V(p_i)$, called the voronoi cell of p_i.

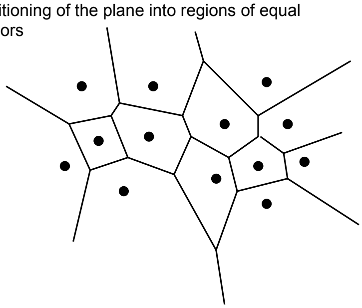
Example

$V(p_i) = \cap_{1 \leq j \leq n,\, j \neq i}\, h(p_i, p_j)$



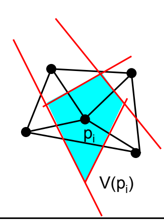
Computing the Voronoi Diagram

Input: A set of points (sites)
Output: A partitioning of the plane into regions of equal nearest neighbors



Animations of the Voronoi diagram

http://wwwpi6.fernuni-hagen.de/java/JavaAnimation

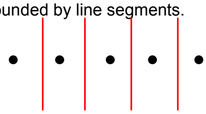


Characteristics of Voronoi Diagrams

1) Voronoi regions (cells) are bounded by line segments.

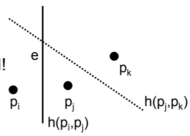
Special case :

Collinear points



Theorem : Let P be a set of n points (sites) in the plane. If all the sites are collinear, then $Vor(P)$ consists of n-1 parallel lines and n cells. Otherwise, $Vor(P)$ is aconnected graph and its edges are either line segments or half-lines.

If p_i, p_j, are not collinear with p_k, then $h(p_i, p_j)$ and $h(p_j, p_k)$ can not be parallel!



Vor(P) is Connected

Claim: *Vor(P)* is connected

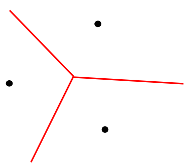
Proof by contradiction:

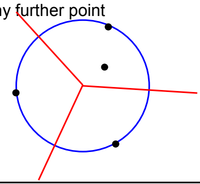
If *Vor(P)* is not connected then there would be a Voronoi cell $V(P_i)$ splitting the plane into two halfes. Beacuse Voronoi cells are convex, $V(P_i)$ would consist of a strip bounded by two parallel full lines, but we know that edges of Voronoi diagram cannot be full lines, hence a contradiction.

Other Characteristics (Assumption: No 4 points are on the circle)

(2) Each vertex (corner) of *VD(P)* has degree 3

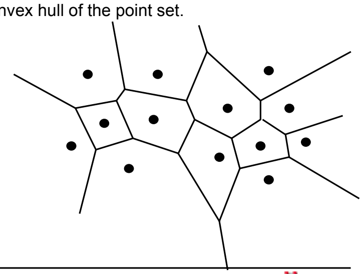
(3) The circle through the three points defining a Vertex of the Voronoi diagram does not contain any further point





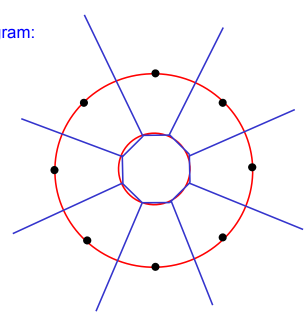
(4) Each nearest neighbor of one point defines an edge of the Voronoi region of the point.

(5) The Voronoi region of a point is unbounded if the point lies exactly on the convex hull of the point set.



Size and Storage

Size of the Voronoi Diagram:



V(p) can have O(n) vertices!

Theorem

The number of vertices in the Voronoi diagram of a set of n points in the plane is *at most 2n-5* and the number of edges is *at most 3n-6*.

Proof: 1. Connect all Half-lines with fictitious point ∞
2. Apply Euler`s formula: $v - e + f = 2$

For $VD(P) + \infty$: v = number of vertices of $VD(P) + 1$
e = number of edges of $VD(P)$
f = number of sites of $VD(P) = n$

Each edge in $VD(P) + \infty$ has exactly two vertices and each vertex
of $VD(P) + \infty$ has at least a degree of 3:

$\Rightarrow$ sum of the degrees of all vertices of $Vor(P) + \infty$

$= 2 \cdot$ (# edges of $VD(P)$)

$\geq 3 \cdot$ (# vertices of $VD(P) + 1$)

Proof(Continued)

Number of vertices of $VD(P) = v_p$

Number of edges of $VD(P) = e_p$

We can apply: $(v_p + 1) - e_p + n = 2$

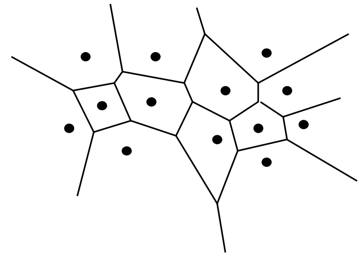
$$2\,e_p \geq 3\,(v_p + 1)$$

$$2\,e_p \geq 3\,(\,2 + e_p - n)$$

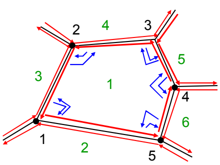
$$= 6 + 3e_p - 3n$$

$$3n - 6 \geq e_p$$

Example



Storage of Voronoi-Diagrams



Three Records:

vertex {
Coordinates
Incident edge
};

face {
OuterComponent
InnerComponents
};

halfedge {
Origin
Twin
IncidentFace
Next
Prev
};

e.g. :

Vertices 1 = {(1,2) | 12}

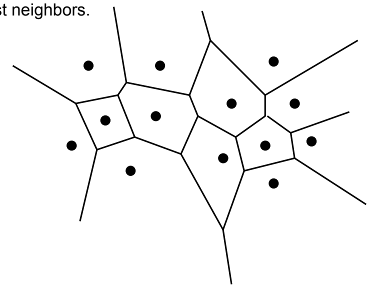
Sites 1 = {15 | [] }

Edges 54 = { 4 | 45 | 1 | 43 | 15 }

Computing the Voronoi Diagram

Input: A set of points (sites)

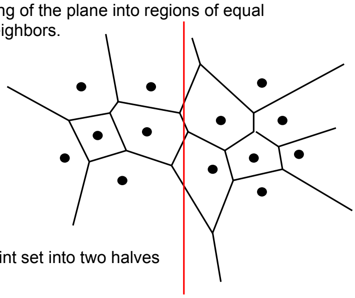
Output: A partitioning of the plane into regions of equal nearest neighbors.



Divide and Conquer(Divide)

Input: A set of points (sites)

Output: A partitioning of the plane into regions of equal nearest neighbors.

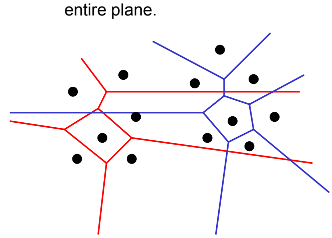


Divide: Divide the point set into two halves

Divide and Conquer (Conquer)

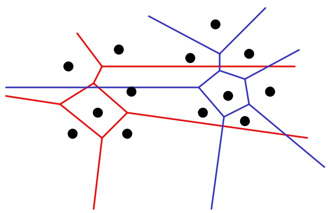
Conquer: Recursively compute the Voronoi diagrams for the smaller point sets

Abort condition: Voronoi diagram of a single point is the entire plane.



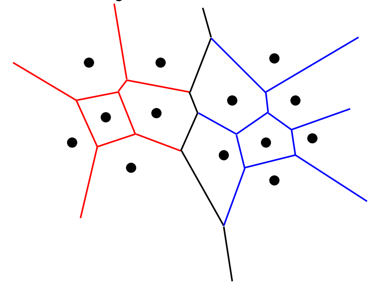
Divide and Conquer (Merge)

Merge the diagrams by a (monoton) sequence of edges)



The Result

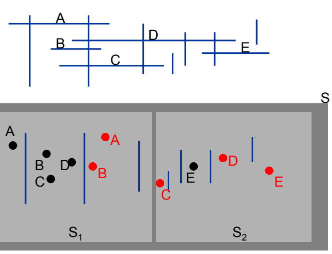
The finished Voronoi Diagram



Running time: With n given points is $O(n \log n)$

Geometrical Divide and Conquer

Problem: Determine all intersecting pairs of segments

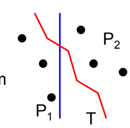


DAC - Construction of the Voronoi diagram

Divide:
Divide P by a vertical dividing line T into 2 equal size subsets say P_1 and P_2. If $|P| = 1 \Rightarrow$ completed.

Conquer:
Compute $VD(p_1)$ and $VD(p_2)$ recursively.

Merge:
Compute the edge course K separating P_1 and P_2
Cut $VD(P_1)$ and $VD(P_2)$ by means of K starting from
Veinige $Vor(P_1)$ and $Vor(P_2)$ and K

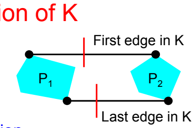


Theorem: K in $O(n) \Rightarrow$ Running time $T(n) = O(n \log n)$

Proof : $T(n) = 2\,T(n/2) + O(n)$, $T(1) = O(1)$

Computation of K

4 tangential points P_1 P_2
Observation: K y - monotonous



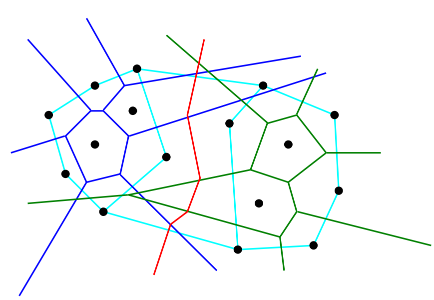
Incremental (sweep line) construction
(p_1 in P_1 and p_2 in P_2 perpendicular with m, Sweep l)

Determines intersection s_1 of m with $Vor(p_1)$ below l
Determines intersection s_2 of m with $Vor(p_2)$ below l

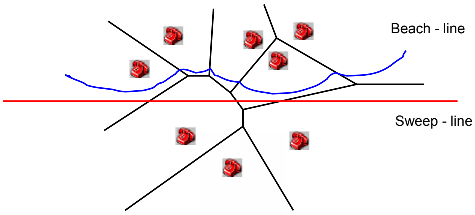
Extend K by line segment l s_i
Set $l = s_i$
Compute new K defining pair p_1, p_2

Theorem : Running time $O(n)$
Proof : $Vor(p_i)$ are convex, therefore each one's forward - edge are only visited once.

Example



Fortune´s Algorithm



Observations:
Intersection of the parabolas define edges
New "telephones" ( ) define new parabolas
Parabola intersection disappear, if *C(P, q)* has 3 points

Use (static object set)

Closest pair of points:
Go through edge list for *VD(P)* and determine minimum

All next neighbors :
Go through edge list for *VD(P)* for all points and get next neighbors in each case

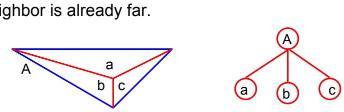
Minimum Spanning tree (after Kruskal)
1. Each point *p* from *P* defines 1-element set of
2. More than a set of *T* exists
   2.1) find *p,p´* with *p* in *T* and *p´* not in *T* with *d(p, p´)* minimum.
   2.2) connect *T* and *p´* contained in *T´* (union)

Theorem : All computes in *O(n log n)*

Applications (dynamic object set)

Search for next neighbor :
Idea : Hierarchical subdivision of *VD(P)*
Step 1 : Triangulation of final Voronoi regions
Step 2 : Summary of triangles and structure of a search tree

Rule of Kirkpatrick :
Remove in each case points with degree < 12,
its neighbor is already far.



Theorem : Using the rule of Kirkpatrick a search tree of logarithmic depth develops.

Duality and Arrangements

- Duality between lines and points
- Computing the level of points in an arrangement
- Arrangements of line segments
- Half-plane discrepancy

Different duality mappings

A point p = (a,b) and a line l: y= mx + b are uniquely determined by two parameters.

a) Slope mapping: p * = L(p): y = ax + b

b) Polar mapping: p *: ax + by = 1

c) Parabola mapping: p*: y=2ax -b

d) Duality transform:
p = (a,b) is mapped to p *: y = ax – b
l: y = mx + b is mapped to l* = (m, -b)

Duality transform

p = (p_x, p_y)

$$(p_x, p_y) \mapsto y = p_x x - p_y$$

$$y = mx + b \mapsto (m, -b)$$

Characteristics :

1. $(p^*)^* = p = (p_x, p_y)$, $(l^*)^* = l$

   $p^*: y = p_x x - p_y$

   $(p^*)^* = (p_x, p_y) = p$

   $(l^*)^* = l$

Characteristics of the duality transform

2) Incidence Preserving :

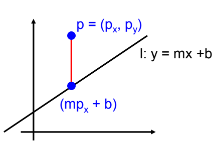
p = (p_x, p_y) lies on l: y = mx+b iff l* lies on p*

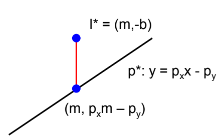
p lies on l iff $p_y = mp_x + b$.

l* lies on p*
iff (m,-b) fulfills the equation $y = p_x x - p_y$
iff $-b = p_x m - p_y$.

Characteristics of the duality transform

3) Order Preserving : p lies above l iff l* lies above p*





p lies above l
$p_y > mp_x + b$

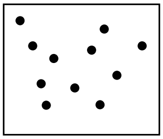
l* lies above p*
$-b > p_x m - p_y$ iff $p_y > p_x m + b$

Summary

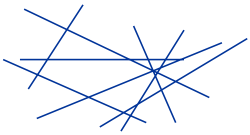
Observations:

1. Point p on straight line l iff point l * on straight line p *
2. p above l iff l * above p *

Computing the level of points in arrangements



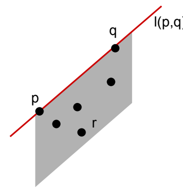
Compute for each pair (p,q) of points
and the straight line l(p,q) defined by p and q:

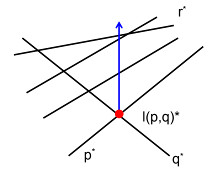


The number of points
- above l(p, q)
- on l(p, q)
- below l(p, q)

⇒ running time (naive):

Determining the number of points below a line



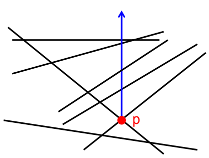


r is below l(p,q) iff l(p,q)* is below r *

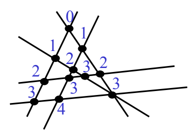
Determining the level of points

Define for a set of straight lines for each intersection point p, the number of those straight lines, which run above p.

Definition: Level of a point p = # straight lines above p



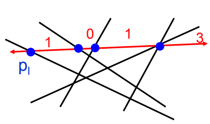
Levels of points in an arrangement



Determining the levels of all Intersections

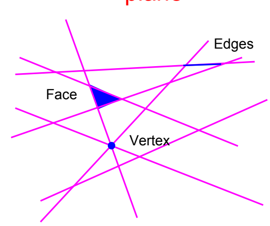
For each straight line:

1) Compute the level the leftmost intersection with other lines in time O(n) (comparison with all other straight lines).
2) Walk along the line and update the level at each intersection point



Run time : $O(n^2)$

Arrangement of a set of n straight lines in the plane



Size of an Arrangement

Theorem :

Let L be a set of n lines in the plane, and let A(L) be the arrangement induced by L.

1) The number of vertices of A(L) is at most n(n-1)/2.
2) The number of edges of A(L) is at most n².
3) The number of faces of A(L) is at most n²/2 + n/2 + 1.

Equality holds in these three statements iff A(L) is simple.

Proof : Assume that A(L) is simple.

1) Any pair of lines gives rise to exactly one vertex
   ⇒ n(n-1)/2 vertices.
2) # of edges lying on a fixed line = 1 + # of intersections on that line with all other lines, which adds up to n.

So total number of edges of A(L) = n².

Proof(Contd...)

Bounding the # of faces

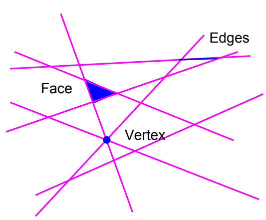
Euler's Formula : For any connected planar embedded graph with m_v veritces, m_e edges, m_f faces the relation $m_v - m_e + m_f = 2$ holds.

We add a vertex v_∞ to A(L) to get a connected planar embedded graph with v vertices, e arcs and f faces.

So we have f = 2 – (v + 1) + e

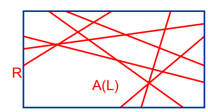
= 2 – (n(n - 1)/2 + 1) + n²

= n²/2 + n/2 + 1.



Storage of an Arrangement

Bounding-box R contains all vertices of A(L).



Store A(L) as doubly connected edge list.

Computation of the Arrangement

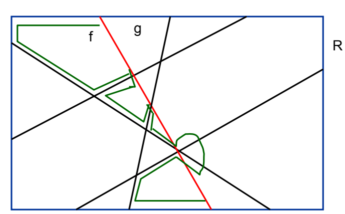
Modify plane-sweep algorithm for segment intersection: Θ(n² log n), there are max. n² intersections.

Incremental algorithm, running in time O(n²)

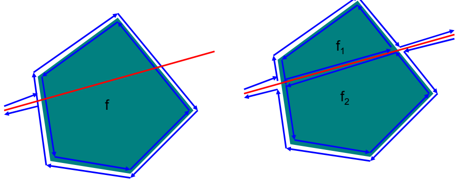
1) Compute Bounding box B(L) that contains all vertices of A(L) in its interior.
2) Construct the doubly connected edge list for the sub-division induced by L on B(L).
3) for i = 1 to n
   1) do find the edge e on B(L) that contains the leftmost intersection point of l_i and A_i.
   2) f = the bounded face incident to e.
   3) while f is not the unbounded face
   4) do split f, and set f to be the next intersected face.

Finding the next intersected face

Idea: Traverse along the edges of faces intersected by g



Splitting a Face

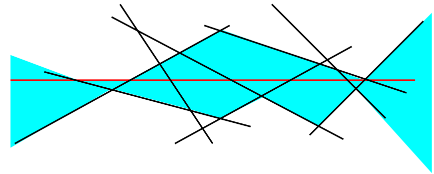


- a new face
- a new vertex
- two new half-edges

Time : O(1)

Zone Theorem

Complexity of the zone of a line : Sum of number of edges and vertices of all intersected faces.

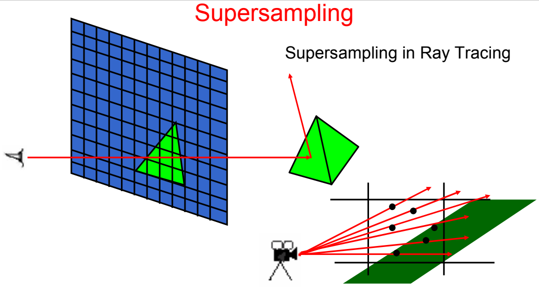


Zone Theorem : The complexity of the zone of a line in an arrangement of m lines in the plane is O(m).

Proof: By induction on m. (Omitted)

Supersampling

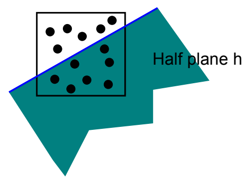
Supersampling in Ray Tracing



In order to handle arbitrary lines: Choose a random set of points (Supersampling):
Shoot many rays through a box, take the average

Computing the Discrepancy

Unit square U
[0,1] x [0,1]



S is set of n sample points in U.

H = set of all halfplanes

Continuous measure of half-plane $h \in H$ is $\mu(h)$

Discrete measure of h is $\mu_s(h)$
$\mu_s(h) := \text{card}(S \cap h) / \text{card}(S)$.

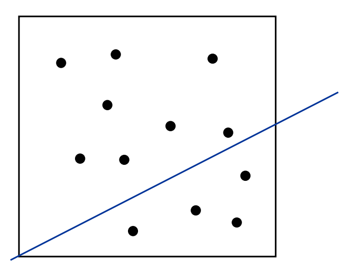
The half-plane discrepancy of a set S of n points is the supremum of all differences between the discrete and continuous measures for all halfplanes.

Definition of half-plane discrepancy

The discrepancy of h with respect to S, denoted as $\Delta_s(h)$, is absolute difference between the continuous and discrete measure. $\Delta_s(h) := | \mu(h) - \mu_s(h) |$.

Halfplane discrepancy : $\Delta_H(s) := \sup_{h \in H} \Delta_s(h)$

Example



Computing the Discrepancy(contd...)

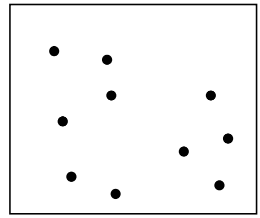
Lemma : Let S be a set of n points in the unit square U.

A half-plane h that achieves the maximum discrepancy with respect to S is of one of the following types :

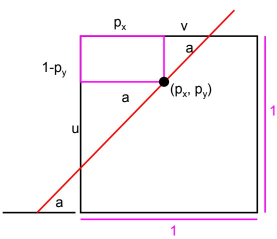
1. h contains one point $p \in S$ on its boundary.
2. h contains 2 or more points of S on its boundary.

The number of type(1) candidates is O(n), and they can be found in O(n) time.

Example



First case : One point



$A(a) = 1/2\ (1 - p_y + p_x \tan a)\ (p_x + (1 - p_y) / \tan a)$

Area function has only finitely many extreme values!

Discussion of the area function

$A(a) = 1/2\ (1 - p_y + p_x \tan a)\ (p_x + (1 - p_y) / \tan a)$

with tan' = 1/cos², (1/x)' = -1/x², chain rule

$\Rightarrow A'(a) = 1/2\ (p_x^2 / \cos^2 a + (1 - p_y)^2/\cos^2 a \tan^2 a)$

$A'(a) = 0 \Rightarrow p_x^2 - (1 - p_y)^2/ \tan^2 a$

$\Rightarrow \tan^2 a = (1 - p_y)^2/p_x^2$

Overview

- Motivation.
- Triangulation of Planar Point Sets.
- Definition and Characterisitics of the Delaunay Triangulation.
- Computing the Delaunay Triangulation
  (randomized, incremental).
- Analysis of Space and Time Requirement.

Motivation

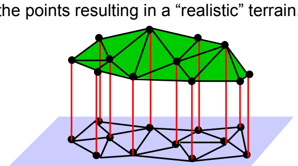
Transformation of a topographic map

into a perspective view

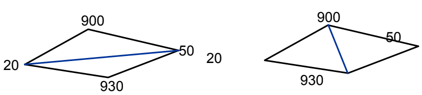
Terrains

Given: A number of sample points $p_1, \ldots, p_n$

Required: A triangulation T of the points resulting in a "realistic" terrain.



"Flipping" of an edge:

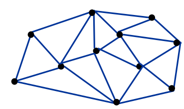


Goal: *Maximise* the minimum angle in the triangulation

Triangulation of Planar Point Sets

Given: Set P of n points in the plane (not all collinear).

A triangulation $T(P)$ of P is a planar subdivision of the convex hull of P into triangles with vertices from P.



$T(P)$ is a maximal planar subdivision.

For a given point set there are only finitely many different triangulations.

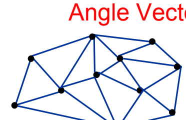
Size of Triangulations

Theorem : Let P be a set of n points in the plane, not all collinear and let k denote the number of points in P that lie on the boundary of convex of hull of P. Then any trianglation of P has $2n-2-k$ triangles and $3n-3-k$ edges.

Proof :

Let T be triangulation of P, and let m denote the # of triangles of T.
Each triangle has 3 edges, and the unbounded face has k edges.
$\Rightarrow n_f$ = # of faces of triangulation = $m + 1$
every edge is incident to exactly 2 faces.
Hence, # of edges $n_e = (3m + k)/2$.
Euler's formula : $n - n_e + n_f = 2$.
Substituting values of n_e and n_f, we obtain:
$m = 2n - 2 - k$ and $n_e = 3n - 3 - k$.

Angle Vector

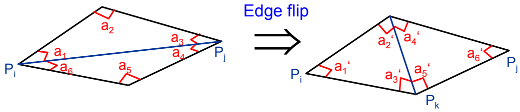


Let $T(P)$ be a triangulation of P (set of n points).
Suppose $T(P)$ has m triangles.
Consider the $3m$ angles of triangles of $T(P)$, sorted by increasing value.
$A(T) = \{ a_1, \ldots, a_{3m} \}$ is called *angle-vector* of T.

Triangulations can be sorted in lexicographical order according to $A(T)$.

A triangulation $T(P)$ is called *angle-optimal* if $A(T(P)) \geq A(T'(P))$ for all triangulations T' of P.

Illegal Edge



The edge $p_i p_j$ is illegal if $\min_{1\le i\le 6} \alpha_i < \min_{1\le i\le 6} \alpha'_i$

Note: Let T be a triangulation with an illegal edge e.
Let T' be the triangulation obtained from T by flipping e.
Then, $A(T') > A(T)$.

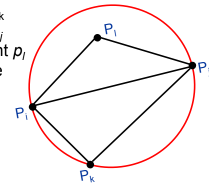
Legal Triangulation

Definition : A triangulation $T(P)$ is called a legal triangulation, if $T(P)$ does not contain any illegal edges.

Test for illegality

Lemma :

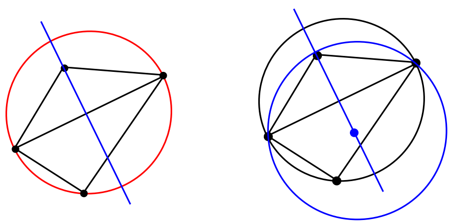
Let edge $\overline{p_i p_j}$ be incident to triangles $p_i p_j p_k$ and $p_i p_j p_l$, and let C be the circle thru p_i, p_j and p_k . The edge $\overline{p_i p_j}$ is illegal *iff* the point p_l lies in the interior of C. Furthermore, if the points p_i, p_j, p_k, p_l form a convex quadrilateral and do not lie on a common circle , then exactly one of $\overline{p_i p_j}$ or $\overline{p_k p_l}$ is an illegal edge.



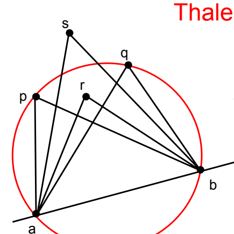
Test of Illegality

Observation:

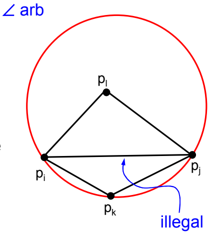
p_l lies inside the circle through p_i, p_j and p_k iff p_k lies inside the circle through p_i, p_j, p_l . When all four points lie on circle, both $\overline{p_i p_j}$ and $\overline{p_k p_l}$ are legal.



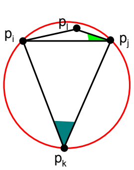
Thales Theorem



∠ asb
< ∠ aqb = ∠ apb
< ∠ arb



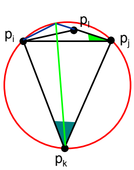
Lemma: Let C be the circle through the triangle p_i, p_j, p_k and let the point p_l be the fourth point of a quadrilateral.
The edge $\overline{p_i p_j}$ is illegal iff p_l lies in the interior of C.



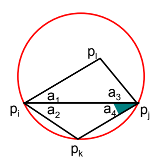
Consider the quadrilateral with p_l in the interior of the circle that goes through p_i, p_j, p_k.

Claim: The minimum angle does not occur at p_k!

(likewise: Minimum angle does not occur at p_l)

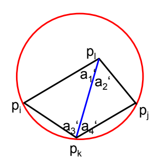


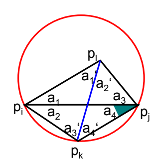
Goal: Show that $\overline{p_i p_j}$ is illegal

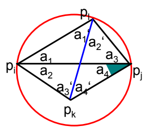


W.l.o.g.

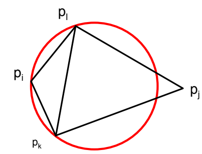
a_4 minimal



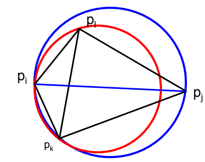




Circle criterion violated ⇒ illegal edge

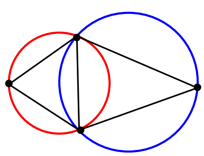


Assumption:
edge $\overline{p_k p_l}$ is illegal,
and circle criterion is not violated



Then: Edge $\overline{p_i p_j}$ is also illegal,
a contradiction!

Circle Criterion



Definition:
A triangulation fulfills the circle criterion *if and only if* the circumcircle of each triangle of the triangulation does not contain any other point in its interior.

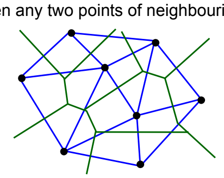
Theorems

Theorem:
A triangulation *T(P)* of a set *P* of points does not contain an illegal edge *if and only if* nowhere the circle criterion is violated.

Theorem:
Every triangulation *T(P)* of a set *P* of points can be finally transformed into an *angle-optimal* triangulation in a finite number of steps.

Definition and characteristics of the Delaunay triangulation

The Delaunay Triangulation *DT(G)* is the straight line dual of the Voronoi diagram.
Vertices: Points (sites) of the Voronoi regions
Edges: Between any two points of neighbouring Voronoi regions

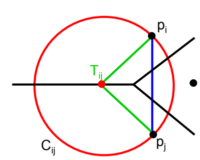


Each Voronoi vertex is the center of a triangle of the Delaunay triangulation (for sets of points (sites) in *general position*).

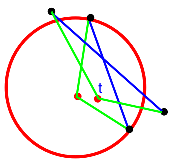
Planarity of the Delaunay Graph DG(P)

Theorem:
The Delaunay Triangulation *DT(P)* of a set of points P is planar.

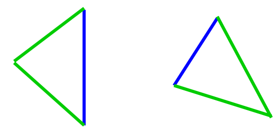
Proof:
Let $p_i p_j$ be an edge of *DT(P)*. Then there is an empty circle C_{ij}, that goes through p_i and p_j.



T_j, the center of C_{ij}, is on the common edge of $V(p_i)$ and $V(p_j)$.



t contains no sites

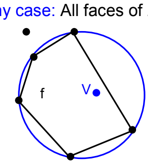


Delaunay Triangulation

A set of points *P* is in *general position* if it contains no 4 points on a circle

For point sets in general position all vertices of the Voronoi diagram have degree 3 and all bounded faces of *DT(P)* are triangles

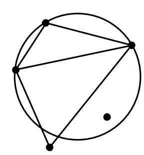
In any case: All faces of *DT(P)* are *convex*

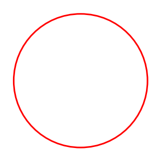


DT(P) = Triangulation of *DG(P)*

Characterisation of the Delaunay Triangulation

Theorem:
Let *P* be a set of points in the plane (in general position), and let *T* be a triangulation of *P*. Then *T* is a Delaunay Triangulation of *P* *if and only if* the circumcircle of any triangle of *T* does not contain any other point of *P* in its interior (i.e. T fulfills the circle criterion).





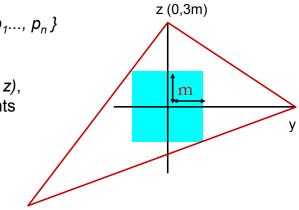
Equivalent characterisations of the Delaunay Triangulation

1. *DT(P)* is the straight-line-dual of *VD(P)*.
2. *DT(P)* is a triangulation of *P* such that all edges are legal (*local angle-optimal*).
3. *DT(P)* is a triangulation of *P* such that for each triangle the circle criterion is fulfilled.
4. *DT(P)* is *global angle-optimal* triangulation.
5. DT(P) is a triangulation of P such that for each edge $\overline{p_i p_j}$ there is a circle, on which p_i and p_j lie and which does not contain any other point from *P*.

Computation of the Delaunay Triangulation (randomized, incremental)

Given: Point set $P = \{p_1, \dots, p_n\}$

Initially:
Compute triangle *(x, y, z)*, which includes the points $p_1, \dots, p_n$.



Algorithm DT(P)

m = max {$|x_i|, |y_i|$}

T = ((3m, 0), (3m, 3m), (0, 3m))

1. initialize *DT(P)* as *T*.
2. permutate the points in *P* randomly.
3. for *r = 1* to *n* do

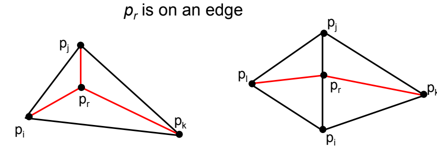
   find the triangle in *DT(P)*, which contains p_r;

   insert new edges in *DT(P)* to p_r;

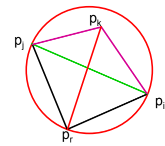
   legalize new edges.
4. remove all edges, which are connected with *x, y* or *z*.

Inserting a point

2 cases : p_r is inside a triangle
p_r is on an edge



Legalize $(p_r, \overline{p_i p_j}, T)$
if $\overline{p_i p_j}$ is illegal
then Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.
Legalize $(p_r, \overline{p_i p_k}, T)$
Legalize $(p_r, \overline{p_k p_j}, T)$



Algorithm Delaunay Triangulation

Input: A set of points $P = \{p_1, \ldots, p_n\}$ in general position

Output: The Delaunay triangulation DT(P) of *P*

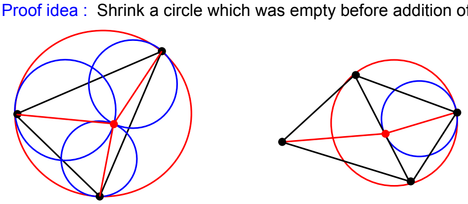
1. *DT(P) = T = (x, y, z)*
2. for *r = 1* to *n* do
3. find a triangle $p_i p_j p_k \in T$, that contains p_r.
4. if p_r lies in the interior of the triangle $p_i p_j p_k$
5. then split $p_i p_j p_k$
6. Legalize($p_r, \overline{p_i p_j}$), Legalize($p_r, \overline{p_i p_k}$), Legalize($p_r, \overline{p_j p_k}$)
7. if p_r lies on an edge of $p_i p_j p_k$ (say $\overline{p_i p_j}$)
8. then split $p_i p_j p_k$ and $p_i p_j p_l$
   Legalize ($p_r, \overline{p_i p_l}$), Legalize ($p_r, \overline{p_i p_k}$),
   Legalize ($p_r, \overline{p_j p_l}$), Legalize ($p_r, \overline{p_j p_k}$)
9. Delete *(x, y, z)* with all incident edges to *P*

Correctness

Lemma :Every new edge created in the algorithm for constructing DT during the intersection of p_r is an edge of the Delaunay graph of $\Omega \cup \{p_1, \ldots, p_r\}$.

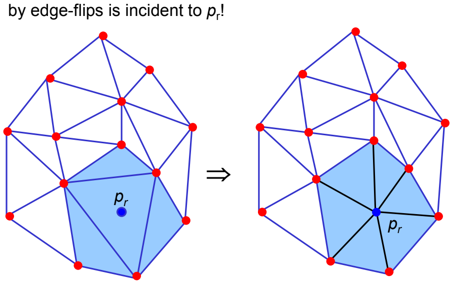
pq is a Delaunay edge *iff* there is a (empty) circle, which contains only *p* and *q* on the circumference.

Proof idea : Shrink a circle which was empty before addition of p_r !



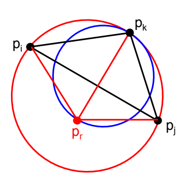
Correctness of the algorithm: Consider newly produced edges:

Observation: After insertion of p_r , every new edge produced by edge-flips is incident to p_r!



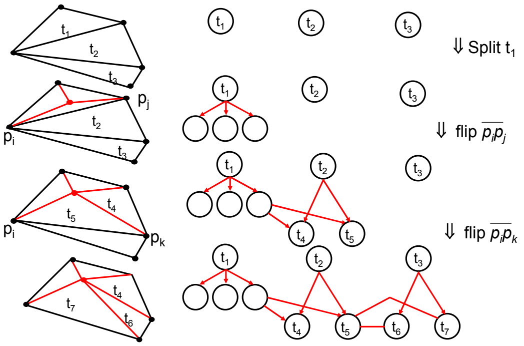
Edge-flips produce only legal edges.

Before inserting p_r , circle that goes through p_i, p_j, p_k was empty!



Edge-flips produce edges that are always incident to p_r !

Data Structure for Point Location



Analysis of the Algorithm for Constructing DT(P).

Lemma :

The expected number of triangles created by the incremental algorithm for constructing DT(P) is atmost $9n + 1$.

Analysis of the Running time

Theorem :

The Delaunay triangulation of a set of P of n points in the plane can be computed in $O(n \log n)$ expected time, using $O(n)$ expected storage.

Proof :

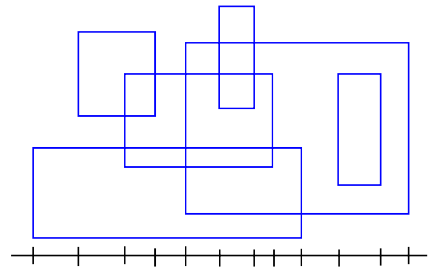
Running time without Point Location :
Proportional to the number of created triangles = $O(n)$.

Point Location :
The time to locate the point p_r in the current triangulation is linear in the number of nodes of D that we visit.

Geometric Data Structures

1. Rectangle Intersection
2. Segment trees
3. Interval trees
4. Priority search trees

Rectangle Intersection



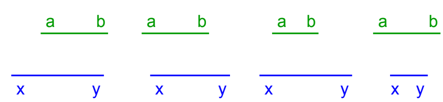
- Sweep a horizontal Scan-Line from top to bottom.
- Store the intersection points with the rectangles in a status structure L.

Operations on L

- Insertion of an interval into L
- deletion of an interval from L
- For a given interval I :
  Determine all intervals from L, *which overlap themselves with I*

L stores a set of intervals over a *discrete* and *well-known* universe of possible end-points.

Reduction of the overlap-query



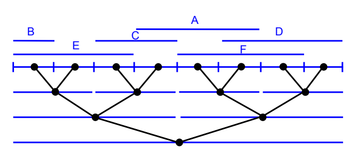
Segment Trees

Segment trees are a structure for storing sets of intervals, which support the following operations:

- insertion of intervals
- deletion of intervals
- stabbing queries:
  For a given point A, report all intervals which contain A (which are stabbed by A)

For the solution of the rectangle intersection problem *semi-dynamic* segment trees are sufficient.

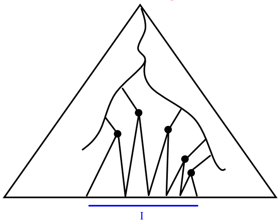
Example



An interval *I* is in the list of a vertex *p if and only if p* is the first node from the root, so that the interval of *I(p)* is contained in *I*.

Insertion of an interval is possible in *O(log n)* steps.

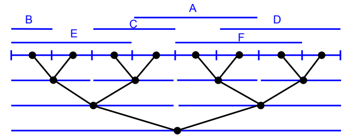
Size of a Segment Tree



Each interval of *I* appears in at the most *O(log n)* interval lists.

Construction of a segment tree with *n* intervals is possible in time *O(n log n).*

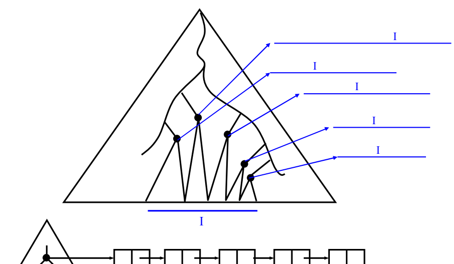
Algorithm for answering stabbing queries



procedure report (p: node ; x: point):
report all intervals of the list of p;
if p is leaf then finish else
{ if (p has left child p_l & x in $I(p_l)$)
then report(p_l, x);
if (p has right child p_r & x in $I(p_r)$)
then report(p_r, x); }

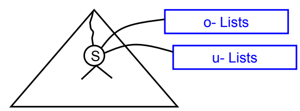
Using the segment tree all intervals that contain a query point can be reported in time *O(log n + k),* where k is the number of reported intervals.

Deletion of Intervals



Dictionary for all intervals

Interval Trees



Skeleton (complete search tree of the interval boundaries)

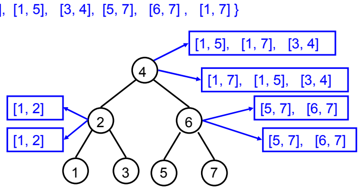
o-Lists sorted according to descending upper end points

u-Lists sorted according to ascending lower end points

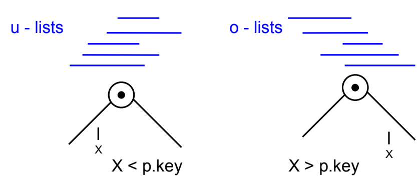
Interval [l,r] is stored in the u-/ o-list of the node s forwards if and only if s of the knots of minimum depth is, so that s lies in [l,r].

Example

{[1, 2], [1, 5], [3, 4], [5, 7], [6, 7] , [1, 7] }



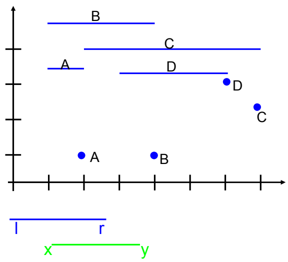
Insertion and deletion of intervals in an interval tree with skeleton of size *O(n)* and altogether *O(n)* intervals can be carried out in time *O(log n).*



Procedure report (p :nodes, x : points)
if x = p.key then report all intervals of the u/o – lists
else if x < p.key then { report beginning of the u-list;
report(p_l, x) }
else (x > p.key) { report beginning of the o - list;
report(p_r, x) }

Stabbing queries can be answered in *O(log n + k)* time, where k is the number of reported intervals.

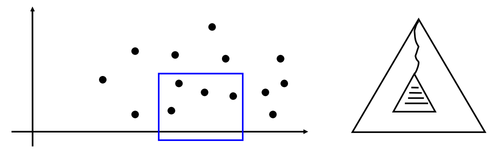
Priority Search Trees



Priority Search Trees

Priority Search trees are a 1.5-dim structure for the storage of points, they support the following operations :

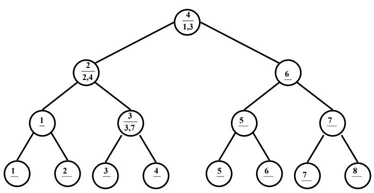
Insertion of a point
Deletion of a point
South-grounded range queries



Priority Search Trees

Priority search trees are
- *binary leaf search trees* for the x-coordinates of the points.
- *min heaps* for the y-coordinates of the points.

M = { (1, 3), (2, 4), (3, 7), (4, 2), (5, 1), (6, 6), (7, 5), (8, 4) }

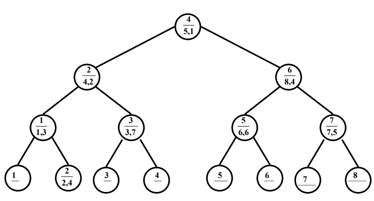


Insertion

Insertion of a point *p = (x, y)* :
Deposite *p* on the search path for *x* according to its *y*-value!
I.e. if on the way down the tree, *p* meets a point *q, with larger y-value*, then deposit p there and and continue the procedure with *q*.

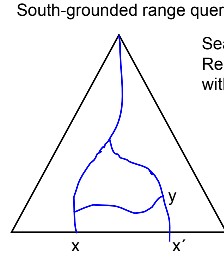
Insertion of a point can be carried out in time *O(log n).*

Deletion



Look for point p in the tree and remove it;
Close the gaps (recursively) by pulling up the point, with smaller y-value.
Deletion of a point is possible in time *O(log n).*

South-grounded range queries (x, x´, y) :

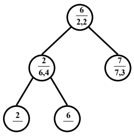


Search for x and x´.
Report all points with y-value < y within the range between these borders.

Executable in *O(log n + k)* time.

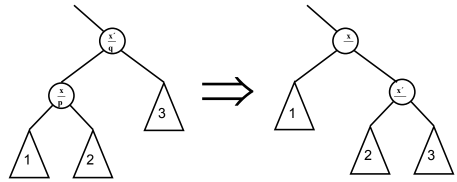
Possibilities for the full dynamization of priority search trees:

No rigid skeleton, but growing or shrinking with the point set.



Inserting (5,3)

Balanced trees as skeletons



Rotation conserves the x-order and destroys in general the y-order.

Special Cases of the Hidden Line Elimination Problem

HLE– Problem :

Produce a realistic image of a given *3- d* scene under orthographic projection by eliminating hidden lines.

***3 - d* Scene :**

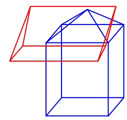
Set of bounding polygonal faces ; each face given by its plane equation and the sequence of its edges ; each edge given by its endpoints.

Special Cases :

Set of

1) *rectilinear faces*
2) *C- oriented faces*

Visibility problems



Hidden-line-elimination

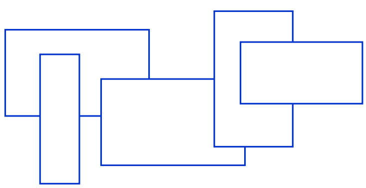
Visible surface computation

Problem Sets

Problem A :

Set of aligned rectangular faces in *3* - space;

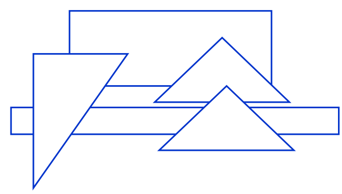
each face parallel to the projection plane.



Problem Sets

Problem B :

Set of *C*- oriented polygonal faces in *3* - space;
all parallel to the projection plane.

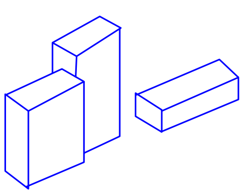


Only C different edge directions

Problem Sets

Problem C :

Set of *C*- oriented solids in *3* – space.



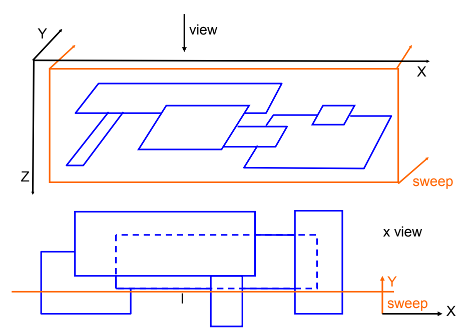
Projection of the faces onto the projection plane yields a set of *C´*- oriented polygons where

$$C' = \binom{C}{2} = O(C^2)$$

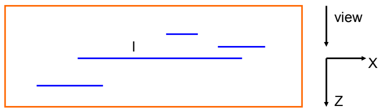
Solution methods:

- *plane- sweep*
- *dynamic contour maintenance*

Plane sweep solution of Problem A

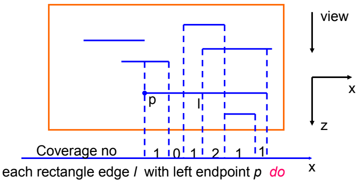


Plane sweep solution of Problem A(contd...)



When sweeping the *X*- *Z*- plane in *Y*- direction:

horizontal line segments
- *appear*
- *stay for a while*
- *disappear*



for each rectangle edge *l* with left endpoint *p* *do*

1. Compute the coverage number *c* of p w.r.t. the currently active faces in front of *p*;
2. Scan along *l* updating *c*; report all pieces with *c = 0* as visible

L = Set of currently active line segments

for each end of a line segment *l´* in *L* passed when scanning along *l* *do*
  if *l´* is above *l* then update c;
    output visible piece, if *c* becomes *0*
  else ignore this end

Subproblems

Subproblem 1:

Given a set *L* of horizontal line segments and a query point *p*, determine the number of segments in *L* that are above *p*.

Subproblem 2:

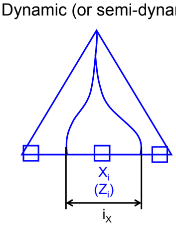
L_X = set of *X*- values of endpoints of segments in *L*

For a given *X*- interval i_X retrieve the coordinates in L_X enclosed by i_X in *X*- order.

L and L_X must allow *insertions* and *deletions*

Solution of Subproblem 2

Dynamic (or semi-dynamic) range tree



r_e = # vertical edges that intersect *l*

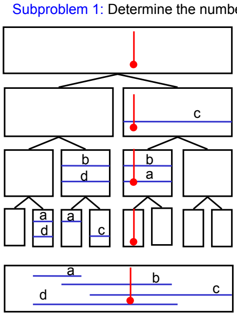
$O(\log n + r_e)$

Above- l- test:

By associated *Z*- values in $O(1)$ time

Segment - range tree

Subproblem 1: Determine the number of segments above a query point



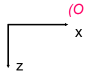
1. Store the *X*- intervals in a segment tree
2. Organize the node lists as range trees according to their *Z*- values

Query interval

Retrieval of the *t* segments in *L* with *Z*- values in | takes time $O(\log^2 n + t)$

We need only the number of those segments

➨ Segment - Rank tree $(O \log^2 n)$



Time Complexity

For each rectangle edge *e* :

$O(\log^2 n)$	for solving *subproblem 1*
$O(\log^2 n + k_e)$	for solving *subproblem 2*
$O(\log^2 n)$	for *inserting/ deleting* a horizontal line segment in a segment range tree
$O(\log n)$	for *inserting/ deleting* two coordinates in a range tree

Space complexity

$O(n \log n)$ for *storing* a segment range tree of *n* elements

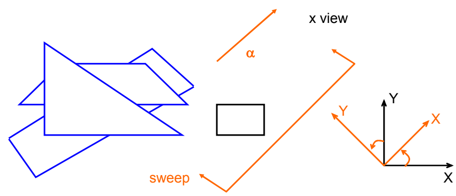
Theorem:

For a set of *n* rectangular faces, *Problem A* can be solved in $O(n \log^2 n + k)$ time and $O(n \log n)$ space, where *k* is the number of edge intersections in the projection plane

Compare with $O((n+k) \log n)$ time!

Problem B

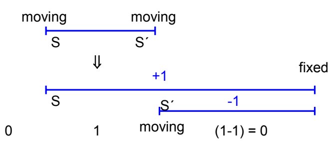
Problem B: *C*- oriented polygonal faces all parellel to the projection plane



Main idea: Use *C* different data structures, one for each edge orientation (speed)

Store moving horizontal objects in a data structure that moves at the same speed as the objects stored in it

Represent horizontal segments by two half-lines



$([x_1,x_2], y) \Rightarrow \begin{cases} ([x_1,\infty], y, +1) \\ ([x_2,\infty], y, -1) \end{cases}$

Subproblem 1: (Determining the number of *segments above query point p*)

For each speed *S* of the *C* possible speeds:
Store the segments with endpoints moving at speed *S* in a *segment rank tree* (associated to *S*)

To obtain the number of segments above *p*:
query all C segment range trees and *add the results*

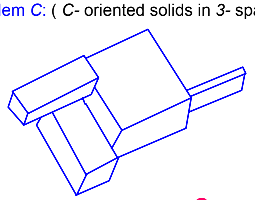
C segment rank trees

Problem B can be solved with the same asymptotic time and space bounds as Problem A

($O(n \log^2 n + k)$ time, $O(n \log n)$ space)

C – Oriented Solids in 3 - Space

Problem *C*: (*C*- oriented solids in *3*- space)



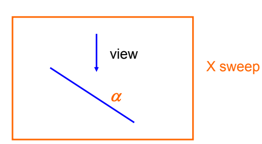
Preprocessing step (requires $O(n)$ time)

1. Compute the set of faces
2. Remove all back faces

C orientations of faces ➨ $C' = \binom{C}{2}$ edge orientations

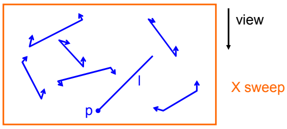
Problem C (contd...)

For each edge-orientation α perform a plane-sweep by choosing a sweep plane which is parellel to α and the direction of view.



Moving Segments in the Sweep Plane

Moving segments in the sweep plane



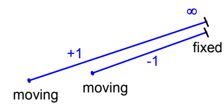
Intersection of the sweep plane with any face is still a line segment having one of C different orientations,

each of its endpoints moves at one of $C' = \binom{C}{2}$ different (speed, direction) pairs

Moving Segments in the Sweep Plane(contd...)

Apply the same solution technique

Represent slanted segments by pairs of slanted half-lines



Solution of Problem C

Same technique as for Problem *B* is applicable.

Solution of Problem *C*: (*C*- oriented solids)

time $O(n \log^2 n + k)$

space $O(n \log n)$

time and space increase with $O(C^3)$

»» *feasible only for small values of C*

Best known algorithm for the general problem

A. Schmitt: time $O(n \log n + k \log n)$

space $O(n + k)$

Output sensitive HLE

n = size of input
k = # edge intersections in projected scene
q = # *visible* edges

large block hiding a complicated scene
»» $k = O(n^2)$, $q = O(1)$

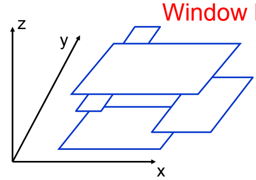
Problem:
Does there exist any algorithm for the *HLE*- problem whose complexity does not depend on *k* but only on *n* and *q* , i.e. on the number of *visible line segments* ?

Problem A (*rect. faces, parellel to proj. plane*)	yes
Problem B (*C- oriented faces, parellel to proj. plane*)	yes
Problem C (*C- oriented solids*)	?

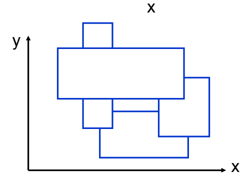
Solution technique:

Dynamic contour maintenance when scanning the objects from front to back

Special Case of HSR / HLE : Window Rendering



Isothetic rectangles in front – to – back order.



Visible portion

Dynamic Contour Maintenance

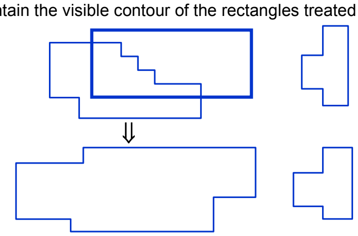
Dynamic contour maintenance

Construct the visible scene by inserting objects from the front to the back into an intially empty scene.

At each stage maintain the contour of the area covered by objects so far. When encountering a new object check it against the current contour to determine its visible pieces and update the contour.

Front to Back Strategy

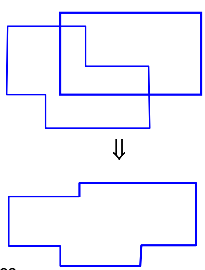
1. Sort the rectangles by increasing depth and treat them in this order
2. Maintain the visible contour of the rectangles treated so far



Compute
1) all intersections of r and c
2) all edges of r completely inside / outside C
3) all edges of C completely inside r

Updating the contour

Case A: Updating the contour



maintain:

E set of contour edges

F set of rectangles whose union is the area within the contour

Algorithm

Algorithm CONTOUR – HLE
Input A set of n rectangular aligned faces *R*, all parellel to the projection plane
Output The set of visible pieces of edges defined by *R*
Method Sort *R* by *z*- coordinates (distance to the observer)

$E := \phi$ { set of contour edges }
$F := \phi$ { set of rectangles whose union is bounded by E }

Scan *R* (according to ascending z-values)

for each rectangle $r \in R$ do
1. Compute all intersections between edges in *r* and edges in *E*

{1a} for each intersected edge $e \in E$ do
delete *e* from *E*;
compute the parts *e* outside *r* insert them into *E*

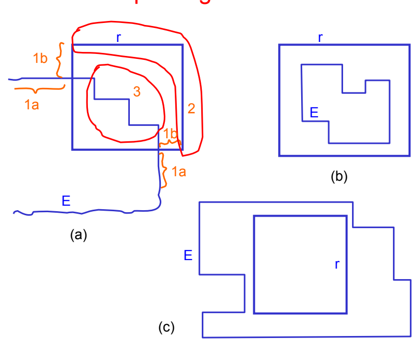
od

Algorithm (contd...)

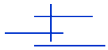
for each edge *e´*of *r* intersecting same edge in *E* do
{1b} compute the pieces of *e´* outside the contour;
report those pieces as visible;
insert those pieces into *E*;
od
2. for each edge *e´* of *r* not intersecting anything do
check *e´* using *F* whether it is completely inside the contour (hidden);
if *e´* is not inside
then report *e´* as visible;
insert *e´* into *E* fi
od
3. Find all edges in *E* that are completely inside *r* and delete them from *E;*

4. Insert *r* into *F*
end CONTOUR - HLE

Updating the contour

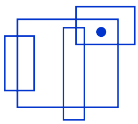


Subproblems



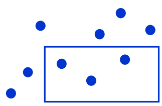
Find intersections between edges in E and r

Segment – Range tree



Given a set of rectangles F and a query point p: check whether p is in UF.

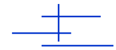
Segment – Segment tree



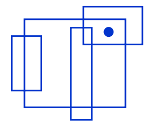
Given a set P of (left end-) points (of edges in E) and a query rectangle r: find all points of P inside r.

Range – Range tree

Sub Problems

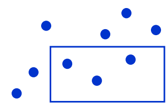


Compute *intersection* between edges in r and C. *segment – range tree* for horizontal , vertical edges of C.



Point – Location in the planar subdivision ∪ C.

segment – segment tree .



Range query for determining all points (representing edges of C) completely inside r. *range – range tree.*

Structures must be dynamic , i.e. Support insert / delete operations efficiently.

Subproblems contd...

Representation of set E of contour edges:

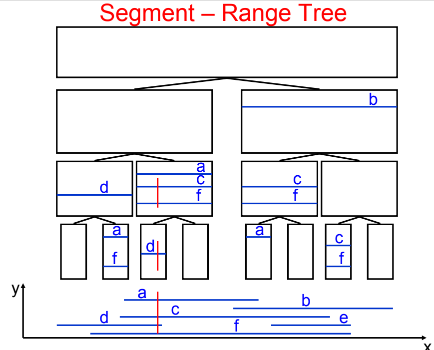
segment – range tree for horizontal edges
segment – range tree for vertical edges
range – range tree for left / bottom end points

Representation of set of rectangles :

segment – segment tree

Update and query take time $O(\log^2 n)$ (+t)

Segment – Range Tree



Reporting all k intersections in time $O(\log^2 n + k)$

Theorem

For each rectangular face r a constant number of operations at a cost $O(\log^2 n)$ per operation is performed. Additional cost arises for each contour edge found as intersecting in step 1 or enclosed in step 3.

Theorem :
For a set of n rectangles, problem A can be solved by dynamic contour maintenance in $O((n + q) \log^2 n)$ time and $O((n + q) \log n)$ space where q is the number of *visible* line segments.

The solution carries over to problem B but not to problem C; because no scanning (´´separation´´) order is defined for problem C.

Theorem(Ottmann / Güting)

Theorem (Ottmann / Güting 1987) :
The window rendering problem for n isothetic rectangles can be solved in time $O((n + k) \log^2 n)$, where k is the size of the output.

Improvements

Bern 1988
$O(n \log n \log\log n + k \log n)$
Preparata / Vitto / Yvinec 1988
$O(n \log^2 n + k \log n)$
Goodrich / Atallah / Overmars 1989
$O(n \log n + k \log n)$ or
$O(n^{1+q} + k)$
Bern 1990
$O((n + k) \log n)$

Can be extended to C – oriented polygons (in depth order)
Problems : 1) arbitrary polygons (in depth order)
2) no depth order