Lecture 4:
(Computational) Geometry

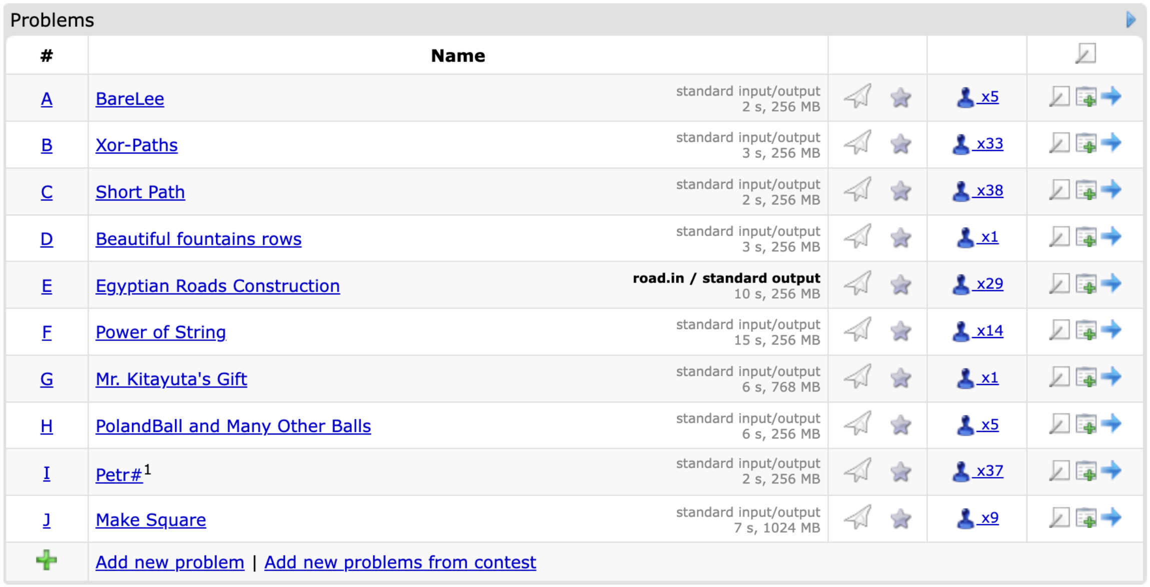
Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Contest Results

Winner: WYWYWYWYWYWYWY (8 problems, 1333 minutes)



Problems

#	Name				
A	BareLee	standard input/output 2 s, 256 MB		x5	
B	Xor-Paths	standard input/output 3 s, 256 MB		x33	
C	Short Path	standard input/output 2 s, 256 MB		x38	
D	Beautiful fountains rows	standard input/output 3 s, 256 MB		x1	
E	Egyptian Roads Construction	**road.in / standard output** 10 s, 256 MB		x29	
F	Power of String	standard input/output 15 s, 256 MB		x14	
G	Mr. Kitayuta's Gift	standard input/output 6 s, 768 MB		x1	
H	PolandBall and Many Other Balls	standard input/output 6 s, 256 MB		x5	
I	Petr#[1]	standard input/output 2 s, 256 MB		x37	
J	Make Square	standard input/output 7 s, 1024 MB		x9	
	Add new problem | Add new problems from contest				

Topic: Geometric Primitives

- Complexities of Geometry
- Fundamental Problems and Algorithms
- Sweepline Algorithms
- Voronoi Diagrams and Delauney Triangulations

Complexities of Geometry

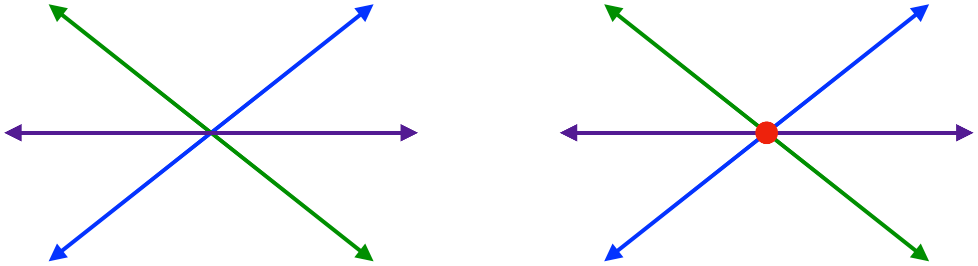
Many programming problems deal with “real” geometry – lines, points, circles, and so forth.

Although you did geometry in high school, it can be surprisingly difficult to program even very simple things. One reason is that floating point arithmetic introduces numerical uncertainty.

Another difficulty of geometric programming is that certain “obvious” operations you do with a pencil, such as finding the intersection of two lines, requires non-trivial programming to do correctly with a computer.

Degeneracy

Special cases or *degeneracies* require extra attention when doing geometric programming. For these reasons I recommend you carefully study my code fragments before writing your own.



Lines

Straight *lines* are the shortest distance between any two points. Lines are of infinite length in both directions, as opposed to *line segments*, which are finite. We limit our discussion here to lines in the plane.

Every line l is completely represented by any pair of points (x_1, y_1) and (x_2, y_2) which lie on it.

Lines are also completely described by equations such as $y = mx + b$, where m is the *slope* of the line and b is the y-*intercept*, i.e., the unique point $(0, b)$ where it crosses the x-axis.

Line Type

Vertical lines cannot be described by such equations, however, because dividing by Δx means dividing by zero. The equation $x = c$ denotes a vertical line that crosses the x-axis at the point $(c, 0)$.
We use the more general formula $ax + by + c = 0$ as the foundation of our line type because it covers all possible lines in the plane:

```
typedef struct {
        double a;               /* x-coefficient */
        double b;               /* y-coefficient */
        double c;               /* constant term */
} line;
```

Multiplying these coefficients by any non-zero constant yields an alternate representation for any line. We establish a canonical representation by insisting that the y-coefficient equal 1 if it is non-zero. Otherwise, we set the x-coefficient to 1:

```
points_to_line(point p1, point p2, line *l)
{
        if (p1[X] == p2[X]) {
                l->a = 1;
                l->b = 0;
                l->c = -p1[X];
        } else {
                l->b = 1;
                l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
                l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
        }
}

point_and_slope_to_line(point p, double m, line *l)
{
        l->a = -m;
        l->b = 1;
        l->c = -((l->a*p[X]) + (l->b*p[Y]));
}
```

Line Intersection

Two distinct lines have one *intersection point* unless they are *parallel*; in which case they have none. Parallel lines share the same slope but have different intercepts and by definition never cross.

```
bool parallelQ(line l1, line l2)
{
     return ( (fabs(l1.a-l2.a) <= EPSILON) &&
              (fabs(l1.b-l2.b) <= EPSILON) );
}
```

The intersection point of lines $l_1 : y = m_1 x + b_1$ and $l_2 : y_2 = m_2 x + b_2$ is the point where they are equal, namely,

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \qquad y = m_1 \frac{b_2 - b_1}{m_1 - m_2} + b_1$$

Implementation

```
intersection_point(line l1, line l2, point p)
{
     if (same_lineQ(l1,l2)) {
         printf("Warning: Identical lines, all points intersect.\n");
         p[X] = p[Y] = 0.0;
         return;
     }

     if (parallelQ(l1,l2) == TRUE) {
         printf("Error: Distinct parallel lines do not intersect.\n");
         return;
     }

     p[X] = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);

     if (fabs(l1.b) > EPSILON)        /* test for vertical line */
             p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
     else
             p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}
```

Angles

An *angle* is the union of two rays sharing a common endpoint. The entire range of angles spans from 0 to 2π radians or, equivalently, 0 to 360 degrees. Most trigonometric libraries assume angles are measured in radians.

A *right* angle measures $90°$ or $\pi/2$ radians.

Any two non-parallel lines intersect each other at a given angle. Lines $l_1 : a_1x + b_1y + c_1 = 0$ and $l_2 : a_2x + b_2y + c_2 = 0$, written in the general form, intersect at the angle θ given by:

$$\tan\theta = \frac{a_1b_2 - a_2b_1}{a_1a_2 + b_1b_2}$$

For lines in slope-intercept form this reduces to $\tan\theta = (m_2 - m_1)/(m_1 m_2 + 1)$.
Two lines are *perpendicular* if they cross at right angles to each other. The line perpendicular to $l : y = mx + b$ is $y = (-1/m)x + b'$, for all values of b'.

Closest Point

A very useful subproblem is identifying the point on line l which is closest to a given point p. This closest point lies on the line through p which is perpendicular to l, and hence can be found using the routines we have already developed:

```
closest_point(point p_in, line l, point p_c)
{
      line perp;               /* perpendicular to l through (x,y) */

      if (fabs(l.b) <= EPSILON) {      /* vertical line */
            p_c[X] = -(l.c);
            p_c[Y] = p_in[Y];
            return;
      }

      if (fabs(l.a) <= EPSILON) {      /* horizontal line */
            p_c[X] = p_in[X];
            p_c[Y] = -(l.c);
            return;
      }

      point_and_slope_to_line(p_in,1/l.a,&perp); /* normal case */
      intersection_point(l,perp,p_c);
}
```

Triangles

Each pair of rays with a common endpoint defines an *internal angle* of a radians and an *external angle* of $2\pi - a$ radians. The three internal (smaller) angles of any triangle add up to $180° = \pi$ radians.

The *Pythagorean theorem* enables us to calculate the length of the third side of any *right* triangle given the length of the other two. Specifically, $|a|^2 + |b|^2 = |c|^2$, where a and b are the two shorter sides, and c is the longest side or *hypotenuse*.

We can go farther to analyze triangles using trigonometry.

Trigonometry

The trigonometric functions *sine* and *cosine* are defined as the x- and y-coordinates of points on the unit circle centered at $(0,0)$. The *tangent* is the ratio of sine over cosine.

These functions enable us to relate the lengths of any two sides of a right triangle T with the non-right angles of T. The non-hypotenuse edges can be labeled as *opposite* or *adjacent* edges in relation to a given angle a. Then

$$\cos(a) = \frac{|\text{adjacent}|}{|\text{hypotenuse}|}, \quad \sin(a) = \frac{|\text{opposite}|}{|\text{hypotenuse}|}, \quad \tan(a) = \frac{|\text{o}}{|\text{a}}$$

Use the famous Indian Chief Soh-Cah-Toa to remember these relations. "Cah" means *C*osine equals *A*djacent over *H*ypotenuse, for example.

Area of a Triangle

The area $A(T)$ of a triangle T is given by $A(T) = (1/2)ab$, where a is the altitude and b is the base of the triangle.

Another approach to computing the area of a triangle is directly from its coordinate representation. Using linear algebra and determinants, it can be shown that the *signed* area $A(T)$ of triangle $T = (a, b, c)$ is

$$2 \cdot A(T) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

This formula generalizes nicely to compute $d!$ times the volume of a simplex in d dimensions.

Note that the signed areas can be negative, so we must take the absolute value to compute the actual area. The sign of this area can be used to build important primitives for computational geometry.

```
double signed_triangle_area(point a, point b, point c)
{
        return( (a[X]*b[Y] - a[Y]*b[X] + a[Y]*c[X]
                - a[X]*c[Y] + b[X]*c[Y] - c[X]*b[Y]) / 2.0 );
}

double triangle_area(point a, point b, point c)
{
        return( fabs(signed_triangle_area(a,b,c)) );
}
```

Circles

A *circle* is defined as the set of points at a given distance (or *radius*) from its *center*, (x_c, y_c). A circle can be represented in two basic ways, either as triples of boundary points or by its center/radius. For most applications, the center/radius representation is most convenient:

```
typedef struct {
        point c;                    /* center of circle */
        double r;                   /* radius of circle */
} circle;
```

The equation of a circle of radius r follows directly from its center/radius representation, $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$. Many important quantities associated with circles are easy to compute. Specifically, $A = \pi r^2$ and $C = 2\pi r$.

A *tangent* line l intersects the boundary of c but not its interior. The point of contact between c and l lies on the line perpendicular to l through the center of c.

We can compute the unknown tangent length x using the Pythagorean theorem. From x, we can compute either the tangent point or the angle a. The distance d from O to the center is computed using the distance formula.

Two circles c_1 and c_2 of distinct radii r_1 and r_2 will intersect if and only if the distance between their centers is at most $r_1 + r_2$.

The points of intersection form triangles with the two centers whose edge lengths are totally determined (r_1, r_2, and the distance between the centers), so the angles and coordinates can be computed as needed.

Line Segments

Computational geometry can be defined (for our purposes) as the geometry of discrete line segments and polygons.

Most computer programs represent geometry as arrangements of line segments. Arbitrary closed curves or shapes can be represented by ordered collections of line segments or *polygons*.

A *line segment* s is the portion of a line l which lies between two given points inclusive. Thus line segments are most naturally represented by pairs of endpoints:

```
typedef struct {
        point p1,p2;            /* endpoints of line segment */
} segment;
```

The most important geometric primitive on segments, testing whether a given pair of them intersect, proves surprisingly complicated because of tricky special cases that arise.

The right way to deal with degeneracy is to base all computation on a small number of carefully crafted geometric primitives. Previously we implemented a general line data type that successfully dealt with vertical lines; those of infinite slope. We can reap the benefits by generalizing our line intersection routines to line segments.

Segment intersection can also be cleanly tested using a primitive to check whether three ordered points turn in a counterclockwise direction.

```c
bool segments_intersect(segment s1, segment s2)
{
   line l1,l2;         /* lines containing the input segments */
   point p;            /* intersection point */

   points_to_line(s1.p1,s1.p2,&l1);
   points_to_line(s2.p1,s2.p2,&l2);

   if (same_lineQ(l1,l2))  /* overlapping or disjoint segments */
           return( point_in_box(s1.p1,s2.p1,s2.p2) ||
                   point_in_box(s1.p2,s2.p1,s2.p2) ||
                   point_in_box(s2.p1,s1.p1,s1.p2) ||
                   point_in_box(s2.p1,s1.p1,s1.p2) );

   if (parallelQ(l1,l2)) return(FALSE);

   intersection_point(l1,l2,p);

   return(point_in_box(p,s1.p1,s1.p2) && point_in_box(p,s2.p1,s2.p2));
}
```

Point in Box

```
bool point_in_box(point p, point b1, point b2)
{
      return( (p[X] >= min(b1[X],b2[X])) && (p[X] <= max(b1[X],b2[X]))
           && (p[Y] >= min(b1[Y],b2[Y])) && (p[Y] <= max(b1[Y],b2[Y])) );
}
```

Polygons and Angle Computations

Polygons are closed chains of non-intersecting line segments. We can implicitly represent polygons by listing the n vertices in order around the boundary of the polygon. Thus a segment exists between the ith and $(i+1)$st points in the chain for $0 \leq i \leq n-1$. These indices are taken mod n to ensure there is an edge between the first and last point:

```
typedef struct {
        int n;                  /* number of points in polygon */
        point p[MAXPOLY];       /* array of points in polygon */
} polygon;
```

Convex Polygons

A polygon P is *convex* if any line segment defined by two points within P lies entirely within P; i.e., there are no notches or bumps such that the segment can exit and re-enter P. This implies that all internal angles in a convex polygon must be *acute*; i.e., at most $180°$ or π radians.

Actually computing the angle defined between three ordered points is a tricky problem. We can avoid the need to know actual angles in most geometric algorithms by using the *counterclockwise predicate* `ccw(a,b,c)`. This routine tests whether point c lies to the right of the directed line which goes from point a to point b.

CCW Predicate / Testing Angle Direction

These predicates are computed using `signed_triangle_area()`. Negative area results if point c is to the left of $\vec{ab}$. Zero area results if all three points are collinear.

```
bool ccw(point a, point b, point c)
{
      double signed_triangle_area();

      return (signed_triangle_area(a,b,c) > EPSILON);
}


bool cw(point a, point b, point c)
{
      double signed_triangle_area();

      return (signed_triangle_area(a,b,c) < EPSILON);
}

bool collinear(point a, point b, point c)
{
      double signed_triangle_area();

      return (fabs(signed_triangle_area(a,b,c)) <= EPSILON);
}
```

Questions?

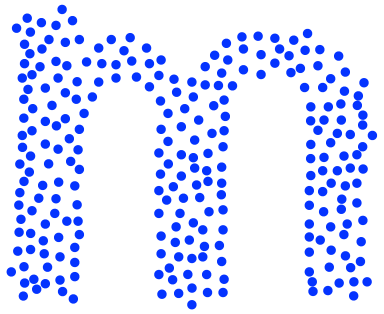
Topic: Fundamental Algorithms and Problems

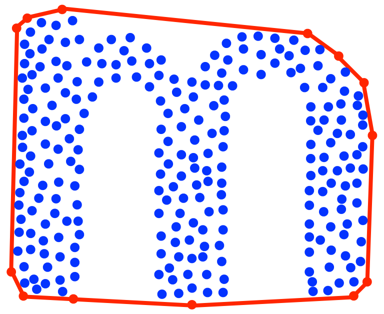
- Complexities of Geometry
- Fundamental Problems and Algorithms
- Sweepline Algorithms
- Voronoi Diagrams and Delauney Triangulations

Convex Hulls

Convex hull is to computational geometry what sorting is to other algorithmic problems, a first step to apply to unstructured data so we can do more interesting things.

The *convex hull* $C(S)$ of a set of points S is the smallest convex polygon containing S.



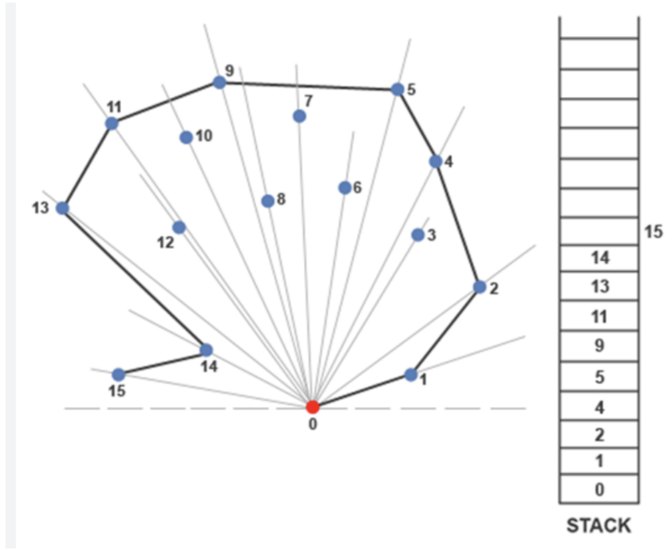


Graham's Scan

The Graham's scan algorithm for convex hull first sorts the points in angular order, and then incrementally inserts the points into the hull in this sorted order. Previous hull points rendered obsolete by the last insertion are then deleted.

Observe that both the leftmost and lowest points *must* lie on the hull, because they cannot lie within some other triangle of points.

Graham Scan: Example



Not removing the contained points connects them all in an ugly but simple polygon.

Analysis of Graham Scan

Sorting by angle takes $O(n \log n)$.

In any iteration, inserting a new point might cause the deletion of $\theta(n)$ points.

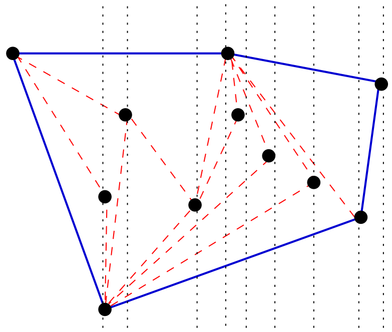
But each of the n points can only be deleted once, so the total running time is linear plus the cost of sorting.

This is a simple example of *amortized analysis*, where the cost is less than n times the worst case.

Graham Scan Implementation

```
point first_point;                  /* first hull point */

convex_hull(point in[], int n, polygon *hull)
{
        int i;                      /* input counter */
        int top;                    /* current hull size */
        bool smaller_angle();

        if (n <= 3) {               /* all points on hull! */
                for (i=0; i<n; i++)
                        copy_point(in[i],hull->p[i]);
                hull->n = n;
                return;
        }

        sort_and_remove_duplicates(in,&n);
        copy_point(in[0],&first_point);

        qsort(&in[1], n-1, sizeof(point), smaller_angle);

        copy_point(first_point,hull->p[0]);
        copy_point(in[1],hull->p[1]);

        copy_point(first_point,in[n]);  /* sentinel for wrap-around */
        top = 1;
        i = 2;

        while (i <= n) {
                if (!ccw(hull->p[top-1], hull->p[top], in[i]))
                        top = top-1;    /* top not on hull */
                else {
                        top = top+1;
                        copy_point(in[i],hull->p[top]);
                        i = i+1;
                }
        }

        hull->n = top;
}
```

Why Did Graham Sort by Angle Instead of x?

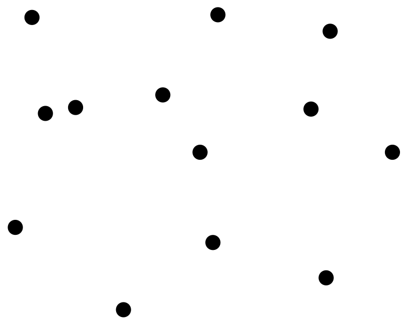


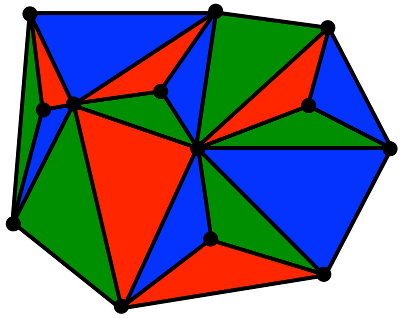
There is no good reason. The incremental insertion algorithm words just as well left-to-right as it does angularly.

The rightmost point must be on the hull, and swallows a chain of points connected to the last point insertion.

Further, left-to-right is more suggestive of the powerful class of sweepline algorithms.

Triangulations





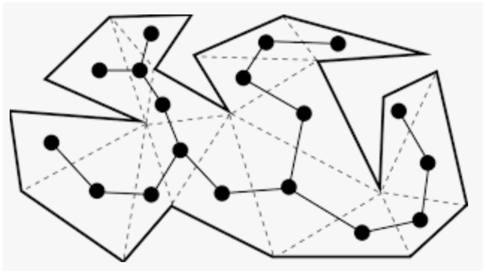
Given a set of n points, we seek to partition its convex hull into empty triangular regions.

How can we construct such a triangulation efficiently?

Triangulation Algorithms

To triangulate a point set, modify the left-to-right Graham Scan to connect the new point to all visible points, instead of deleting them.

van Gogh's Algorithm



Every polygon contains at least two ears, meaning three consecutive points p_{i-1}, p_i, p_{i+1} such that the segment (p_{i-1}, p_{i+1}) do not intersect any part of the polygon.
Van Gogh's algorithm for polygon triangulation works by finding ears and cutting them off.

Area of a Polygon

We can compute the area of any triangulated polygon by summing the area of all triangles.

An even slicker algorithm is based on the notion of signed areas for triangles, which we used in our `ccw` routine. By properly summing the signed areas of the triangles defined by an arbitrary point p with each segment of polygon P we get the area of P, because the negatively signed triangles cancel the area outside the polygon.

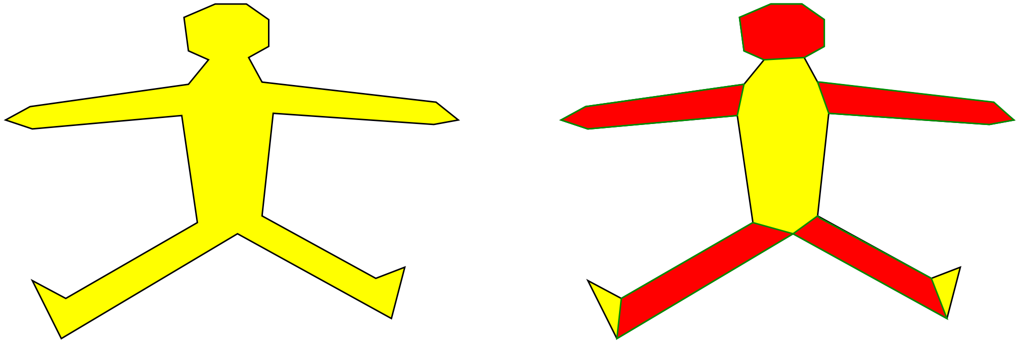
This computation simplifies to the equation

$$A(P) = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

where all indices are taken modulo the number of vertices.

```c
double area(polygon *p)
{
        double total = 0.0;                     /* total area so far */
        int i, j;                               /* counters */

        for (i=0; i<p->n; i++) {
             j = (i+1) % p->n;
             total += (p->p[i][X]*p->p[j][Y]) - (p->p[j][X]*p->p[i][Y]);
        }

        return(total / 2.0);
}
```

Polygon Partitioning



How can you partition a polygon into the smallest number of convex pieces?

Polygon Partitioning Algorithm

Use dynamic programming.
Let $M[i, j]$ be the smallest number of convex pieces the subchain from vertices i to j after adding a chord (i, j).

$$M[i, j] = 1 + \min_{k}(M[i, k], M[k, j])$$

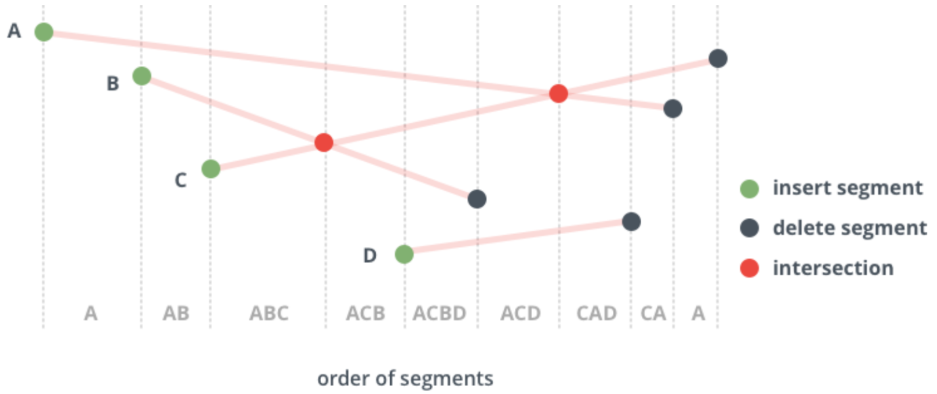
Legal k have no concave angles between for all x between i or j and k.

Questions?

Topic: Sweepline Algorithms and Duality

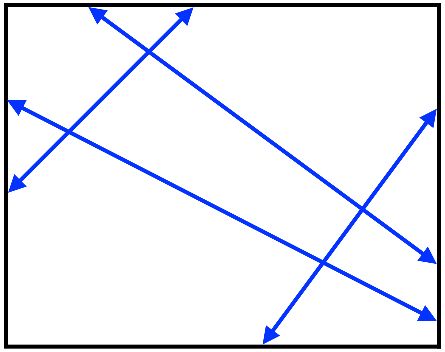
- Complexities of Geometry
- Fundamental Problems and Algorithms
- Sweepline Algorithms
- Voronoi Diagrams and Delauney Triangulations

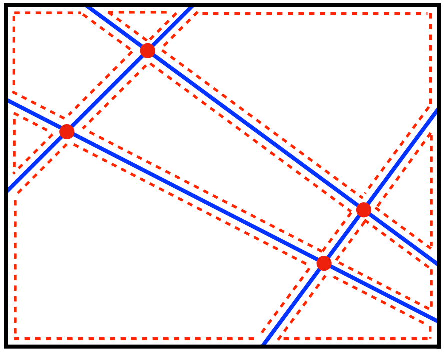
Sweepline Algorithms



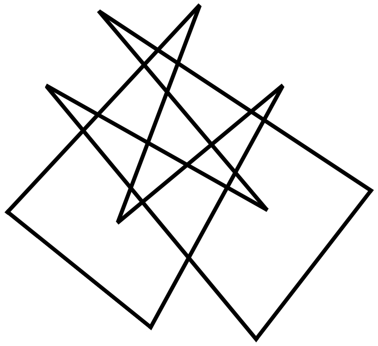
- Need a priority queue to maintain interesting future events.
- Need a horizon data structure to maintain the order of objects hit by the current sweep line position.

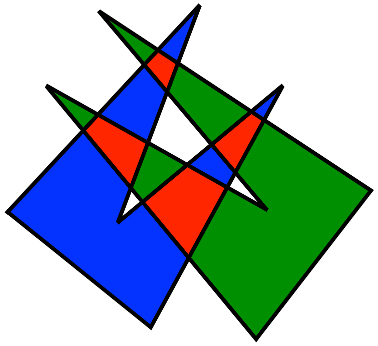
Sweeping Arrangements





Intersection Detection

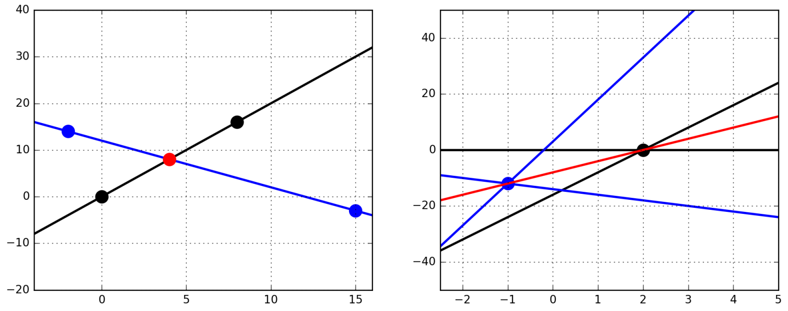




Sweep a line from left to right, stopping at every point of intersection, and make local updates to what is in and what is out.

Duality

In solving linear systems, given n lines we seek the point that lies on all the lines.
In regression, we seek the line that lies on all n points.
By the duality transformation $(s, t) <-> y = (s)x - t$ lines are equivalent to points in another space.

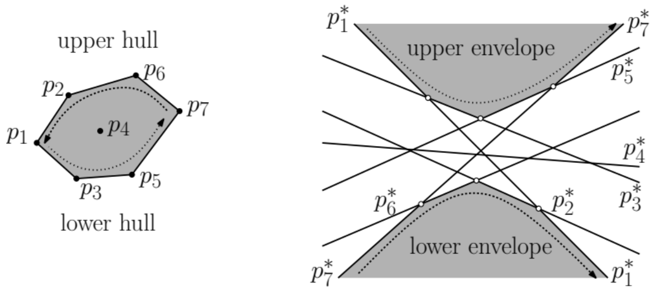


Thinking in Dual Space

It is often useful to view problems in dual space as another way to think about them.

Convex hull finds the smallest region containing all points

Linear programming (half-plane intersection) asks for the largest region on the right side of every linear constraint.



Identifying Geometric Degeneracies

Which is easier:

- Testing whether a set of n points have three points on a line?
- Test whether a set of n lines have three lines going through a single point?

Same Thing

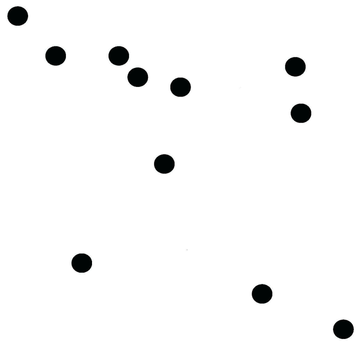
By duality, both of these problems are exactly the same.
But it seems easier to me to test whether there is a point on three lines using a sweepline algorithm.

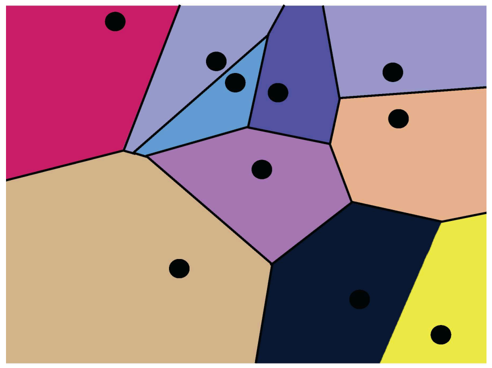
Questions?

Topic: Voronoi Diagrams and Delauney Triangulations

- Complexities of Geometry
- Fundamental Problems and Algorithms
- Sweepline Algorithms
- Voronoi Diagrams and Delauney Triangulations

Voronoi Diagrams

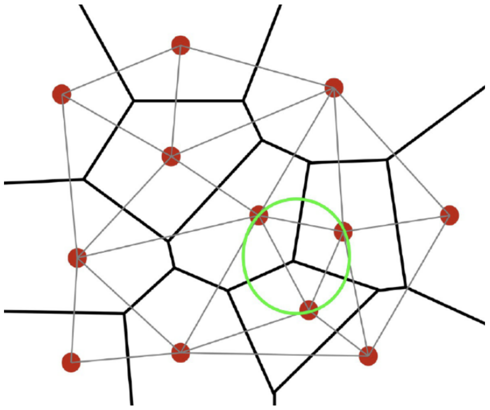




Voronoi diagrams partition the plane around each site p, defining the region where p is the closest site.

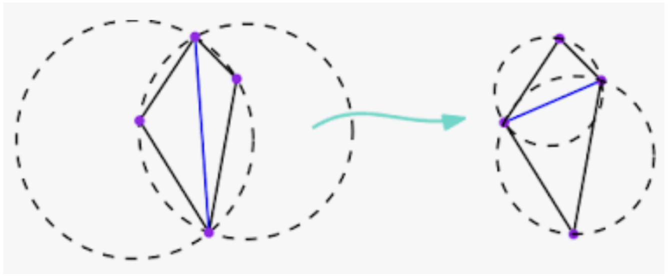
Constructing Voronoi Diagrams

The edges of Voronoi diagrams are constructed from pieces of the perpendicular bisectors of pairs of points.
But the better way to construct a Voronoi Diagram is as a dual graph of the *Delauney triangulation*.



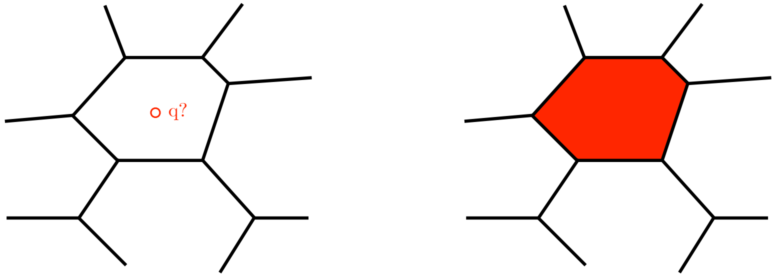
Delauney Triangulations

The key property of a Delauney triangulation is that it has fat triangles, by maximizing the minimum angle over all triangulations.
The Delauney triangulation can be constructed by finding edges to flip that remove small angles.

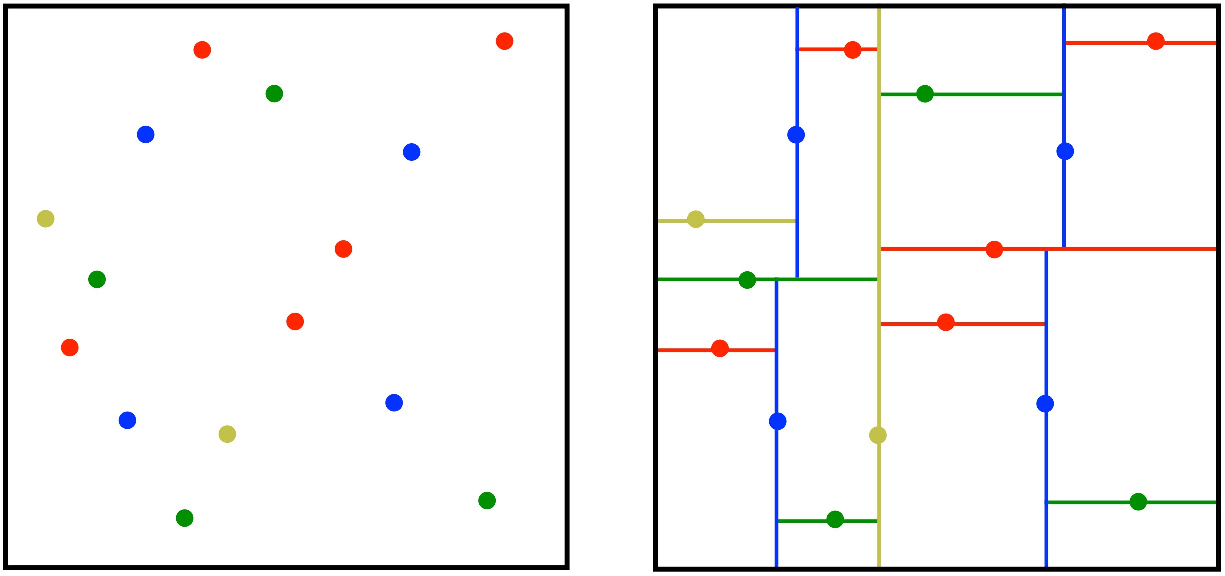


Point Location

The point location problem asks us to determine whether a point is inside or outside a given polygon, or to identify which one of set of polygons the point is in.
One approach is to triangulate the polygon, then test if there exists a triangle containing the point.



KD-trees



For Further Reading

- M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry: Theory and Applications*, second edition, 2000.
- Arseniy Akopyan, *Geometry in Figures*, 2017.

Questions?