GB21802 - Programming Challenges

Week 8 - Computational Geometry

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-06-17,20

Last updated June 20, 2016

Last Week Results

Week 6 - Graph II

- Division – 16/31
- What Base is This? – 6/31
- Divisibility of Factors – 11/31
- Triangle Counting – 11/31
- Help my Brother (II) – 4/31
- Marbles – 1/31
- Ocean Deep! Make it Shallow! – 9/31
- Winning Streak – 0/31

- 14 people: 0 problems;
- 6 people: 1-2 problems;
- 6 people: 3-4 problems;
- 4 people: 5-6 problems;
- 1 people: 7-8 problems!

Special Notes – Grade Relaxation

The following rule will relax the grades requisites a bit:

- Base grade C: You need 1 problem each week. **ALSO, every two problems above the minimum "fix" a week with 0 problems**.
- Base grade B: You need 2 problems each week. **Also, every two problems above the minimum "fix" a week with 1 problem**.
- Base grade A: You need 3 problems each week. **No change in requirement.**

Example

- 3 0 2 2 0: Grade: C
- 3 1 2 2 3: Grade: B
- 4 2 3 3 4: Grade: B

Topic of the Week – Computational Geometry

- Computational Geometry problems are generally considered to be difficult, both in terms of understanding the solution, and programming the solution;
- One trick for these problems is to prepare a large library of basic geometric operations (distances, intersections, angle operations, etc);
  - Focus of this class is the implementation of these operations.
- Special attention is needed to deal with degeneracies;

Degeneracies: Special cases

Two types of degeneracies: Special cases and Precision errors

(some) Special cases:

- Lines parallel to the vertical axis
- Colinear Lines
- Overlapping Segments
- Concave polygons
- Etc...

Good implementations should deal with common special cases.

Degeneracies: Precision errors

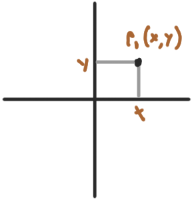
Representation of floating point numbers in computers has a limited precision. So for multiple operations on very small numbers, we may start to see calculation errors.

Some ways to avoid floating point precision errors:

- Whenever possible, convert the float numbers to integers
- Never compare “float x == float y”.
- Instead, use this: “fabs(x - y) < EPS” (float EPS$=$ 0.00000001)

Point Representation

Points are the building blocks of geometric objects. In C/C++, we can represent them using a struct with two members:




```
// When possible, use int coordinates
struct point_i { int x, y;
  point_i() { x = y = 0; }
  point_i(int _x, int _y) : x(_x), y(_y) {}};

// Floating point variation
struct point { double x, y;
  point() { x = y =0.0;}
  point(double _x, double _y) : x(_x), y(_y) {}};
```

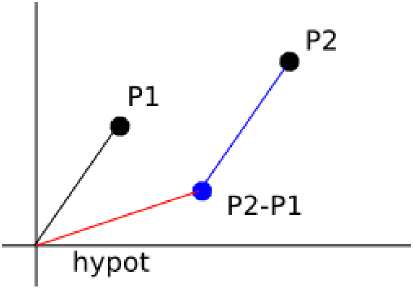
Point Operations

To compare two points, or test for equality, we can overload the *equal* or *less* operator in the point struct.

```
struct point { double x, y;
   point() { x = y = 0.0;
   point(double _x, double _y) : x(_x), y(_y) {}

   // override less than operator -- useful for sorting
   bool operator < (point other) const {
      if (fabs(x - other.x) > EPS)
         return x < other.x;
      return y < other.y; }

   // override equal operator, takes EPS into account
   bool operator == (point other) const {
      return (fabs(x - other.x) < EPS &&
              (fabs(y - other.y) < EPS)); }
   }
```

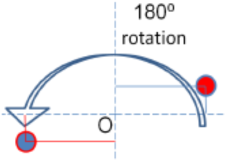
Point: Euclidean Distance

```
#define hypot(dx,dy) sqrt(dx*dx + dy*dy)

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y);
}
```



Point: Rotation around origin

```
#define PI            3.14159265358979323846  /* pi */
#define DEG_to_RAD(X) (X*PI)/180.0

// theta is in degrees
point rotate(point p, double theta) {
   double rad = DEG_to_RAD(theta);
   return point(p.x * cos(rad) - p.y * sin(rad),
                p.x * sin(rad) + p.y * cos(rad));}
```



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Line Representation

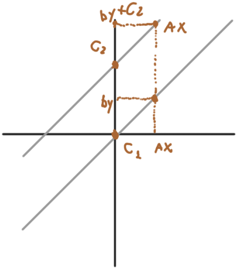
How to represent a line?

- Two points. Problem: cannot generalize for other points of the line easily;
- $y = mx + c$. Problem: cannot handle vertical lines (m is infinite)
- $ax + by + c = 0$. Better representation for “most” cases.

```
struct line { double a,b,c; };

void pointsToLine(point p1, point p2, line &l) {
   if (fabs(p1.x - p2.x) < EPS {
      l.a = 1.0; l.b = 0.0; l.c = -p1.x; }
   else {
      l.a = -(double) (p1.y-p2.y)/(p1.x-p2.x);
      l.b = 1.0; l.c = -(double) (l.a*p1.x) - p1.y;}
}
```

Line: Parallel and Identical lines

- Two lines are parallel if their coefficients (a, b) are the same;
- Two lines are identical if all coefficients (a, b, c) are the same;
- Remember that we force b to be 0 or 1;




```
bool areParallel(line l1, line l2) {
   return (fabs(l1.a-l2.a) < EPS) &&
          (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {
   return areParallel(l1,l2) &&
          (fabs(l1.c - l2.c) < EPS); }
```

Line: Intersection

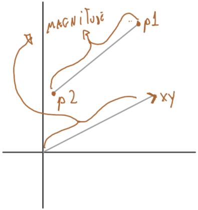
If two lines are not parallel, then they will intersect at a point. This point (x,y) is found by solving the system of two linear equations:

$$a_1x + b_1y + c_1 = 0 \text{ and } a_2x + b_2y + c_2 = 0$$

```
bool areIntersect(line l1, line l2, point &p) {
   if (areParallel(l1,l2)) return False;

   p.x = (l2.b * l1.c - l1.b * l2.c) /
         (l2.a * l1.b - l1.a * l2.b);
   if (fabs(l1.b) > EPS) // Testing for vertical case
      p.y = -(l1.a * p.x + l1.c);
   else
      p.y = -(l2.a * p.x + l2.c);
   return true; }}
```

Segments and Vectors

- A Line Segment is a line limited by two points and finite length;
- A Vector is a segment with an associated direction;
- Often vectors are represented by a single point (the other assumed to be the origin);

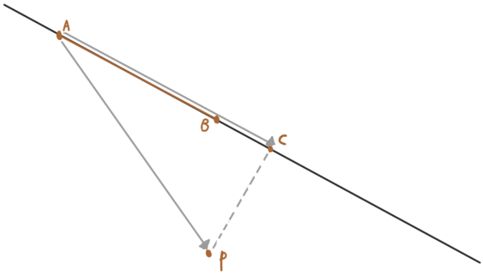



```
struct vec { double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {
    return vec(b.x - a.x, b.y - a.y); }
vec scale(vec v, double s) {
    return vec(v.x * s, v.y * s); }
point translate(point p, vec v) {
    return point(p.x + v.x , p.y + v.y); }
```

Distance between point and line

Given a point p and a line l, the distance between the point and the line is the distance between p and the c, the closest point in l to p.

We can calculate the position of c by taking the projection of $\bar{a}c$ into l (a, b are points in l).



Distance between point and line

```
double dot(vec a, vec b) {
   return (a.x * b.x + a.y * b.y); }
double norm_sq(vec v) {
   return v.x * v.x + v.y * v.y; }

// Calculates distance of p from line, given
// a,b different points in the line.
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
  // translate a to c
  return dist(p, c); }
```

Distance between segment and line

If we have a segment *ab* instead of a line, the procedure to calculate the distance is similar, but we need to test if the intersection point falls in the segment.

```
double distToLineSegment(point p, point a,
                         point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);

  if (u < 0.0) { c = point(a.x, a.y); // closer to a
                 return dist(p, a); }
  if (u > 1.0) { c = point(b.x, b.y); // closer to b
                 return dist(p, b); }

  return distToLine(p, a, b, c); }
```

Angles between segments

angle between two segments ao and ob

```
#import <cmath>

double angle(point a, point o, point b) { // in radians
vec oa = toVector(o, a), ob = toVector(o, b);
return acos(dot(oa, ob)/sqrt(norm_sq(oa)*norm_sq(ob)));}
```

Left/Right test: We can calculate the position of point p in relation to a line l using the cross product.

Take q, r points in l. Magnitude of the cross product pq x pr being positive/zero/negative means that $p \rightarrow q \rightarrow r$ is a left turn/collinear/right turn.

```
double cross(vec a, vec b) {
  return a.x * b.y - a.y * b.x; }
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }
collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
```

Problem Example: UVA – Intersection

Summary

Given two points p_1 and p_2, and a rectangle, test whether the segment $p_1 p_2$ intersects the rectangle.

Strategy

Problem Example: UVA – Intersection

Summary

Given two points p_1 and p_2, and a rectangle, test whether the segment p_1p_2 intersects the rectangle.

Strategy

- Test if points p_1 or p_2 are in the rectangle (easy tests first)
- Test if p_1p_2 intersects with any side of the rectangle.
- "Hard" Way:

Problem Example: UVA – Intersection

Summary

Given two points p_1 and p_2, and a rectangle, test whether the segment $p_1 p_2$ intersects the rectangle.

Strategy

- Test if points p_1 or p_2 are in the rectangle (easy tests first)
- Test if $p_1 p_2$ intersects with any side of the rectangle.
- “Hard” Way:
  - Find the intersection between lines $p_1 p_2$, and top/bottom/left/right
  - Test if the intersection point is in line $p_1 p_2$;
  - Test if the intersection point is in the rectangle;

Problem Example: UVA – Intersection

Summary

Given two points p_1 and p_2, and a rectangle, test whether the segment p_1p_2 intersects the rectangle.

Strategy

- Test if points p_1 or p_2 are in the rectangle (easy tests first)
- Test if p_1p_2 intersects with any side of the rectangle.
- "Hard" Way:
  - Find the intersection between lines p_1p_2, and top/bottom/left/right
  - Test if the intersection point is in line p_1p_2;
  - Test if the intersection point is in the rectangle;
- There is an easier way that takes into account vertical/horizontal sides

Problem Example: UVA – Waterfalls

Summary

Given a list of water sources, and a list of segments, calculate the position that each water source will arrive at the bottom.

Strategy:

- For each water source, calculate all the segments that intersect it (easy because vertical line)
- For each segment, calculate the intersection point - get the highest one.
- New position of the water source is the lowest point of that segment.

Problem Example: UVA – Waterfalls

Summary

Given a list of water sources, and a list of segments, calculate the position that each water source will arrive at the bottom.

Strategy:

- For each water source, calculate all the segments that intersect it (easy because vertical line)
- For each segment, calculate the intersection point - get the highest one.
- New position of the water source is the lowest point of that segment.

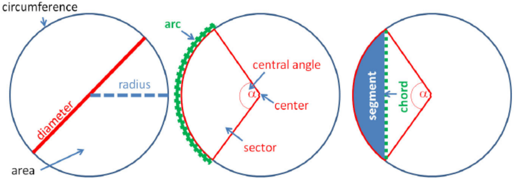
- Problem: No limit of segments or water sources. How do you avoid TLE?

Circles

- A circle is defined by its center (a, b) an its radius r
- The circle contains all points such (x, y) such as $(x-a)^2 + (y-b)^2 \leq r^2$

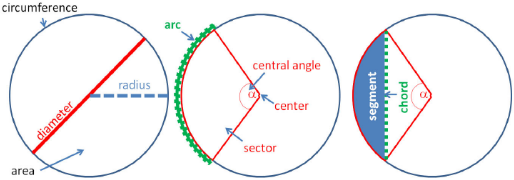
```
int insideCircle(point_i p, point_i c, int r) {
   int dx = p.x-c.x, dy = p.y-c.y;
   int Euc = dx*dx + dy*dy, rSq = r*r;
   return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
   // 0 - inside, 1 - border, 2- outside
}
```

Circles (2)



- If you are not given π, use $pi = 2*\mathrm{acos}(0.0)$;
- Diameter: $D = 2r$; Perimeter/Circumference: $C = 2\pi r$; Area: $A = \pi r^2$;
- To calculat the Arc of an angle α (in Degrees), $\frac{\alpha}{360} * C$;

Circles (3)



- A chord of a circle is a segment composed of two points in the circle's border. A circle with radius r and angle α degrees has a chord of length sqrt($2r^2(1 - \cos\alpha)$)
- A Sector is the area of the circle that is enclosed by two radius and and arc between them. Area is: $\frac{\alpha}{360}A$
- A Segment is the region enclosed by a chord and an arc.

Problem Example: Area

Summary

Given 4 circles, determine the proportion of points that fall in all four circles.

Triangle Basics

Any 2 dimensional polygon can be expressed as a combination of triangles. So triangles are important constructs in computational geometry.

Common Characteristics

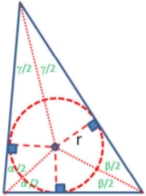
- Triangle Inequality: Sides a, b, c obey $a + b > c$
- Triangle Area: Be b one side of the triangle and h its height, $A = 0.5bh$
- Perimeter: $p = a + b + c$
- Semiperimeter: $s = 0.5p$

Heron's Formula

We can calculate the area of a triangle based on its sides:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

Incircle Triangle



Radius of the Incircle: $r = \text{area}(\Delta)/s$

```
def radiusInCircle(p1,p2,p3):
   ab, bc, cd = dist(p1,p2),dist(p2,p3),
                      dist(p3,p1)
   A = area(ab,bc,ca) % Heron's formula
   P = ab+bc+ca
   return A/(0.5*P)
```

Finding the center point of the Incircle

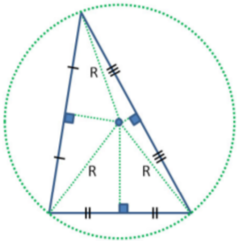
- Check that the three points are not colinear;
- Find the bisection *AP* of the *AB*-*AC* angle;
  - Calculate the point *P* in *BC* that bisects *A*
  - The proportion of *BP* is $(AB/AC)/(1 + AB/AC)$
- Find the bisection BP' of the *BA*-*BC* angle;
- Fint the intersection of AP-BP'

Incircle Triangle

Calculating the Center (Code)

```
int inCircle(point p1, point p2, point p3,
                point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0; // colinear points;
  line l1, l2; // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3),
                      ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);
  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3),
                ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);
  areIntersect(l1, l2, ctr);
  return 1; }
```

Excircle Triangle



Radius of the excircle

A triangle with sides a, b, c and area A has an excircle with radius: $R = abc/4A$.

The center of the excircle is the intersection of the *perpendicular bisectors*.

Trigonometry

- Law of Cosines:
  $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$
  $\gamma = \text{acos}((a^2 + b^2 - c^2/2ab)$
- Law of Sines: (R is the radius of the excircle):
  $a/\sin(\alpha) = b/\sin(\beta) = c/\sin(\gamma) = R$

Example: UVA 11909 - Soya milk

Problem Description

Given the dimensions of a milk box and its inclination, calculate the amount of milk left in the box.

Example: UVA 10577 - Bounding Box

Given three vertices of a regular polygon, calculate the minimal square necessary to cover the polygon.

Hint: You don't actually need to calculate any polygons

Polygons

Definition

A polygon is a plane figure bounded by a finite sequence of line segments.

Polygon Representation

- In general we want to sort the points in CW or CCW order
- Adding the first point at the end of the array helps avoid special cases;

```
// 6 points, entered in counter clockwise order;
vector<point> P;
P.push_back(point(1, 1)); // P0
P.push_back(point(3, 3)); // P1
P.push_back(point(9, 1)); // P2
P.push_back(point(12, 4)); // P3
P.push_back(point(9, 7)); // P4
P.push_back(point(1, 7)); // P5
P.push_back(P[0]); // important: loop back
```

Polygon Algorithms

Perimeter of a Poligon – sum of distances

```
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)
     // remember: P[0] = P[P.size()-1]
     result += dist(P[i], P[i+1]);
  return result; }
```

Area of a Poligon – half the determinant of the XY matrix

```
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1); }
  return fabs(result) / 2.0; }
```

Polygon – Concave and Convex check

Convex Polygons

Has NO line segment with ends inside itself that intersects its edges.

Another definition is that all inside angles “turn” the same way.

Testing for a convex polygon

```
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false; // Not a polygon
  bool isLeft = ccw(P[0], P[1], P[2]); //described earlier
  for (int i = 1; i < sz-1; i++)
    if (ccw(P[i],P[i+1],P[(i+2)==sz? 1 : i+2])!=isLeft)
      return false; // works for both left and right
      // different sign -> this polygon is concave
  return true; }
```

Polygon – Testing Inside or outside

There are many ways to test if a point P is in a polygon.

- Winding Algorithm: Sum the angles of all angles APB (A, B) are points in the polygon. If the sum is 2π. Point is in polygon.
- Ray Casting Algorithm: Draw an segment from P to infinity, and count the number of polygon edges crossed. Odds: Inside. Even: Outside.

Winding Algorithm Code

```
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]); //left turn/ccw
      else sum -= angle(P[i], pt, P[i+1]); } //right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }
```

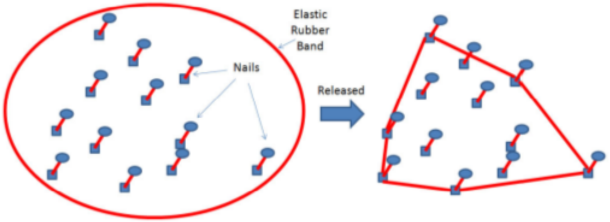
Polygon – Cutting

To cut *P* along a line *AB*, we separate the points in *P* to the left and right of the line.

```
point lineIntersectSeg(point p, point q, point A, point B) {
  double a=B.y-A.y; double b=A.x-B.x; double c=B.x*A.y-A.x*B.y;
  double u=fabs(a*p.x+b*p.y+c); double v=fabs(a*q.x+b*q.y+c);
  return point((p.x*v + q.x*u)/(u+v),
               (p.y*v + q.y*u)/(u+v)); }

vector<point> cutPolygon(point a, point b, const vector<point> &Q){
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1)
      left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS)
      P.push_back(Q[i]); //Q[i] is on the left of ab
    if (left1*left2 < -EPS) //edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b)); }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front()); // make P's first point = P's last point
  return P; }
```

Polygon – Convex Hull

Given a set of points S, the convex hull is the polygon P composed of a subset of S so that every point of S is either part of P, or inside it.



The main algorithm for calculating the convex hull is *Graham's Scan*.

It's idea is to test each point angle order, to see if the point belongs to the hull.

Polygon – Graham's Scan (1)

```
point pivot(0, 0);

bool angleCmp(point a, point b) { // angle-sorting
  if (collinear(pivot, a, b)) // special case
    return dist(pivot, a) < dist(pivot, b);
  // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }

vector<point> CH(vector<point> P) {
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0]==P[n-1])) P.push_back(P[0]); // special case
    return P; }
  // first, find P0 = point with lowest Y and, if tied, righmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y ||
        (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];
  // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);
```

Polygon – Graham's Scan (2)

```
// third, the ccw tests
vector<point> S;
S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);
// initial S
i = 2;
// then, we check the rest
while (i < n) {
  // note: N must be >= 3 for this method to work
  j = (int)S.size()-1;
  if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);
  // left turn, accept
  else S.pop_back(); }
  // or pop the top of S until we have a left turn
return S; }
```

Problem Discussion

- Sunny Mountains
- Bright Lights
- Rope Crisis in Ropeland
- Bounding Box
- Soya Milk
- SCUD Bursters
- Trash Removal
- The Sultan's Problem

Class Summary

Computational Geometry

- Basic Concepts
- Triangles
- Circles
- Polygons

Final Week: String Problems!