Computational Geometry

Migael Strydom

Introduction

Computational geometry problems occur quite frequently in Olympiad problems. It is essential to know at least some of the basics before going to any major programming competition. In addition to what is written here it will be useful to remember all high school geometry and to make use of common sense.

Points and Lines

A point is easily represented as a coordinate pair. In some cases it is necessary to use floating-point variables. However, to avoid the problems associated with floating-point variables, these algorithms will mainly make use of integer coordinates.

Although there are several different ways to represent lines, I will represent them as a pair of points.

CCW

The first and most important operation that is essential to many geometric solutions is the counter-clockwise function. Given the coordinates of 3 points, this function returns 1 if the points are in counter-clockwise order, -1 if they are in clockwise order, and 0 if they are collinear.

```
int ccw(CPoint a, CPoint b, CPoint c)
{
    int dx1 = b.x - a.x;
    int dx2 = c.x - a.x;
    int dy1 = b.y - a.y;
    int dy2 = c.y - a.y;

    if(dx1*dy2 > dy1*dx2)
        return 1;

    if(dx1*dy2 < dy1*dx2)
        return -1;

    return 0;
}
```

This is done by simply comparing the gradients of the lines formed between the first and the second point and between the first and the third point. To avoid a division by 0, you can multiply through by dx1*dx2.

Line Segment Intersection

It is now easy to test for the intersection of two line segments. If both endpoints of each line are on the on different sides of the other, then the line segments must intersect:

```
bool intersect(CLine l1, CLine l2)
{
    return ((ccw(l1.p1, l1.p2, l2.p1)*ccw(l1.p1, l1.p2, l2.p2) < 0)
        &&  (ccw(l2.p1, l2.p2, l1.p1)*ccw(l2.p1, l2.p2, l1.p2) < 0));
}
```

This function, however, does not check for the case where the endpoint of one line segment lies on the other, or if the line segments lie on top of each other. It is important to note that when working with geometry there are often many different special cases that need to be tested for, as in this situation. Whether these special cases are possible with the given input data or need to be tested for depends on the problem requirements.

The following code completes the above `intersect` function.

```
    if(ccw(l1.p1, l1.p2, l2.p1) == 0)
        return point_in_box(l1.p1, l1.p2, l2.p1);

    if(ccw(l1.p1, l1.p2, l2.p2) == 0)
        return point_in_box(l1.p1, l1.p2, l2.p2);

    if(ccw(l2.p1, l2.p2, l1.p1) == 0)
        return point_in_box(l2.p1, l2.p2, l1.p1);

    if(ccw(l2.p1, l2.p2, l1.p2) == 0)
        return point_in_box(l2.p1, l2.p2, l1.p2);
```

Where the `point_in_box` function is:

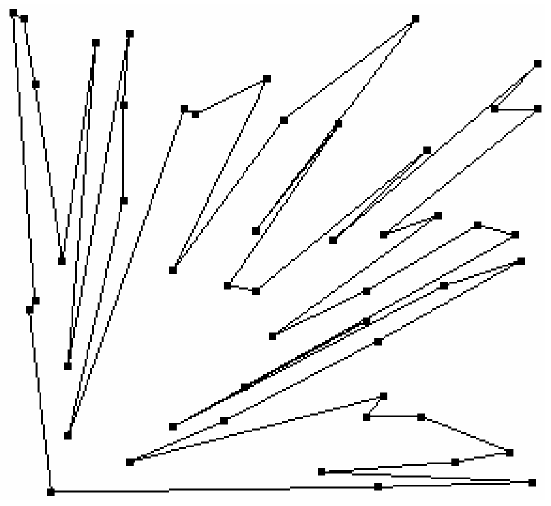
```
bool point_in_box(CPoint corner1, CPoint corner2, CPoint point)
{
    int x1 = min(corner1.x, corner2.x);
    int x2 = max(corner1.x, corner2.x);
    int y1 = min(corner1.y, corner2.y);
    int y2 = max(corner1.y, corner2.y);

    return (point.x >= x1 && point.x <= x2 &&
            point.y >= y1 && point.y <= y2);
}
```

Simple Closed Path

Given a set of *N* random points on a plane it is sometimes necessary to find a path through all of the points that does not intersect itself and returns to the point at which it started. Such a path is called a *simple closed path*. This is an elementary problem because it asks for any such path, unlike the *traveling salesman problem*, which asks for the best path.

A simple way to solve this problem is to choose an anchor point, such as the bottom most point, choosing the left most of all the points that share the same y-value. This point can be considered the first point. All the other points are then sorted according to the angle made between the point, the anchor, and the horizontal line through the anchor. This method is slow, however, because angles need to be calculated. It can be


A simple closed path through random points

made much faster by instead, in sorting, comparing the points directly and not the angles. This is where the `ccw` function becomes handy again.

```
CPoint anchor;
bool compare_points(CPoint a, CPoint b)
{
    int is_ccw = ccw(anchor, a, b);
    if(is_ccw == 1)
        return true;

    if(is_ccw == -1)
        return false;

    //check distances
    return ((anchor.x-a.x)*(anchor.x-a.x) +
            (anchor.y-a.y)*(anchor.y-a.y)) <
           ((anchor.x-b.x)*(anchor.x-b.x) +
            (anchor.y-b.y)*(anchor.y-b.y));
}
```

Notice that, if the points are collinear, they are sorted according to the distance from the anchor point. The reason for this will become clear later.

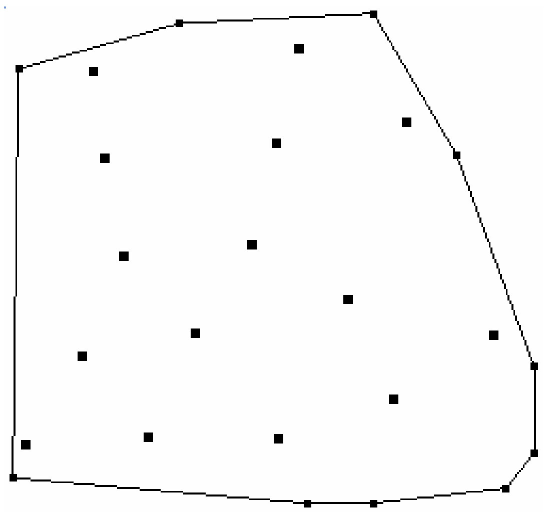
Finding the simple closed path is now a simple matter of sorting, using the given comparison function.

```
CPoint points[NumPoints];
void simple_closed_path()
{
    anchor = points[bottomleft()];
    sort(points, &points[NumPoints], compare_points);
}
```

Convex Hull

The convex hull of a set of points is defined to be the smallest convex polygon that contains all the points. Equivalently, it is the shortest path surrounding all the points. It is easy to prove that the vertices of the convex hull are points from the original point set. The most natural way of thinking about the convex hull is to imagine a set of posts randomly placed in a field, and you walk around them with a rope and tie it to itself after walking all the way around. If the rope is tight around the posts, it forms the convex hull.

The convex hull of a set of points can be found intuitively as follows: Find a point known to be on the convex hull, such as the one with the smallest *y* coordinate. Imagine starting with a horizontal line through this first point that rotates upwards around this point until it hits another point. This next point has to be on the hull. The process can be repeated from this point, each time finding another point on the


The convex hull around a set of points

hull until it gets back to the original point. Now consider the procedure for finding the simple closed path that was described previously. When starting with an anchor that is on the convex hull, clearly the next point in the simple closed path and the last point in the path have to be on the hull as well. Thus 2 new points on the hull have been found. Finding the simple closed path using the next point on the hull as an anchor will produce the next point. This is a simple method for finding the convex hull, although it is very slow. The next section will describe a faster method.

The Graham Scan

The Graham Scan was invented by R L Graham in 1972. It starts, as above, by finding a point from the given set known to be on the convex hull. It then finds the simple closed path through all the points, using this starting point as the anchor. From here on, once the sorting has been done, the Graham Scan runs in linear time. Start "walking" down the path in a counter-clockwise direction, assuming all the points you come to are on the convex hull. When walking counter-clockwise along a convex polygon, you should only be turning left. Therefore, when you come to a point where you have to turn right to get to the next point, you know that the point you are on could not possibly be on the convex hull. You therefore remove all the most recent points from your list of points on the convex hull until you get to a point at which you are not required to turn right to proceed. It is better described by the following algorithm:

```
int grahamscan()
{
    int M, i;
    CPoint t;
    simple_closed_path();

    //sentinal value at the end
    points[NumPoints] = points[0];
    M = 2;

    for(i = 3; i < NumPoints; i++)
    {
        while(ccw(points[M], points[M-1], points[i]) >= 0)
            M = M-1;

        M = M+1;

        //swap
        t = points[M]; points[M] = points[i]; points[i] = t;
    }
    M = M+1;
    return M;
}
```

After calling this function, the first *M* values in the array of points will contain the points forming the vertices of the convex hull in order, starting at the anchor. The rest of the interior points will be in a random order in the rest of the array. Notice that this function returns the value of *M*.

Point Location

Another common problem in computational geometry is that of determining whether a given point lies inside a certain polygon. The easiest method of solving this problem is by imagining a line starting at the point and extending in some direction to infinity. By counting the number of intersections between the line and edges of the polygon, the location of the point can be found. If the line crossed the edges of the polygon an even number of times, the point must be outside the boundaries of the polygon. Otherwise, the point lies on the inside. This problem can be simplified computationally by choosing a horizontal or vertical line to stretch from the point to infinity.

Area Computations
A formula worth remembering is the following:

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

It returns the area of any polygon, given its vertices. *n* is the number of vertices.

It can be implemented as follows:

```
double area()
{
    double total = 0.0;
    int i, j;

    for(i = 0; i < NumPoints; i++)
    {
        j = (i+1) % NumPoints;
        total += (points[i].x*points[j].y) –
                 (points[j].x*points[i].y);
    }

    return total / 2;
}
```