CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

Geometric Algorithms

Potentially **large** set of objects to manipulate.

- Possibly millions of points, lines, squares, circles.
- Efficiency is crucial.

Computational Geometry

- Will concentrate on discrete algorithms – 2D

Practical considerations

- Special cases
- Numeric stability

Same principles often apply to 3D, but it may be more complicated.
We will avoid continuous problems such as polygon intersection.

Special cases: Geometric programming is much like other programming in this sense. But there are a LOT of special cases! Co-point, co-linear, co-planar, horizontal, vertical, etc.

Numeric stability: Each intersection point in a cascade of intersections might require increasing precision to represent the computed intersection, even when the point coordinates start as integers. Floating point causes problems!

Definitions

- A **point** is represented by a pair of coordinates (x, y).
- A **line** is represented by distinct points p and q.
  - Manber's notation: $-p-q-$.
- A **line segment** is also represented by a pair of distinct points: the endpoints.
  - Notation: $p-q$.
- A **path** P is a sequence of points $p_1, p_2, \cdots, p_n$ and the line segments $p_1-p_2, p_2-p_3, \cdots, p_{n-1}-p_n$ connecting them.
- A **closed path** has $p_1 = p_n$. This is also called a **polygon**.
  - Points $\equiv$ vertices.
  - A polygon is a **sequence** of points, not a **set**.

Line alternate representation: slope and intercept.
For polygons, order matters. A left-handed and right-handed triangle are not the same even if they occupy the same space.

Definitions (cont)

- **Simple Polygon**: The corresponding path does not intersect itself.
  - A simple polygon encloses a region of the plane INSIDE the polygon.
- Basic operations, assumed to be computed in constant time:
  - Determine intersection point of two line segments.
  - Determine which side of a line that a point lies on.
  - Determine the distance between two points.

no notes

Point in Polygon

Problem: Given a simple polygon P and a point q, determine whether q is inside or outside P.

Basic approach:
- Cast a ray from q to outside P. Call this L.
- Count the number of intersections between L and the edges of P.
- If count is even, then q is outside. Else, q is inside.

Problems:
- How to find intersections?
- Accuracy of calculations.
- Special cases.

Special cases:
- Line intersects polygon at a vertex, goes in to out.
- Line intersects poly. at inflection point (stays in or stays out).
- Line intersects polygon through a line.

Simplify calculations by making line horizontal.

Accuracy of calculations is not a problem with integer coordinates for points and a horizontal line. But think about representing the intersection point for two arbitrary line segements (from a polygon intersection operation). Cascading intersections can lead to ever-increasing demand for precision in coordinate representation.

Point in Polygon Analysis (1)

Time complexity:
- Compare the ray to each edge.
- Each intersection takes constant time.
- Running time is $O(n)$.

Improving efficiency:
- $O(n)$ is best possible for problem as stated.
- Many lines are "obviously" not intersected.

no notes

Point in Polygon Analysis (2)

Two general principles for geometrical and graphical algorithms:
1. Operational (constant time) improvements:
   - Only do full calculation for 'good' candidates
   - Perform 'fast checks' to eliminate edges.
   - Ex: If $p_1.y > q.y$ and $p_2.y > q.y$ then don't bother to do full intersection calculation.
2. When doing many point-in-polygon operations, preprocessing may be worthwhile.
   - Ex: Sort edges by min and max y values.
     Only check for edges covering y value of point q.

Spatial data structures can help.

"Fast checks" take time. When they "win" (they rule something out), they save time. When they "lose" (they fail to rule something out) they add extra time. So they have to "win" often enough so that the time savings outweighs the cost of the check.

Constructing Simple Polygons

Problem: Given a set of points, connect them with a simple closed path.

Approaches:
1. Randomly select points.
2. Use a scan line:
   - Sort points by y value.
   - Connect in sorted order.
3. Sort points, but instead of by y value, sort by angle with respect to the vertical line passing through some point.
   - Simplifying assumption: The scan line hits one point at a time.
   - Do a rotating scan through points, connecting as you go.

(1) Could easily yield an intersection.

(2) The problem is connecting point p_n back to p_1. This could yield an intersection.

Simplifying assumption is that the points are not colinear w.r.t. the scan line.

See Manber Figure 8.6.

Validation

Theorem: Connecting points in the order in which they are encountered by the rotating scan line creates a simple polygon.

Proof:

- Denote the points $p_1, \cdots, p_n$ by the order in which they are encountered by the scan line.
- For all i, $1 \leq i < n$, edge $p_i - p_{i+1}$ is in a distinct slice of the circle formed by a rotation of the scan line.
- Thus, edge $p_i - p_{i+1}$ does not intersect any other edge.
- Exception: If the angle between points p_i and p_{i+1} is greater than 180○.

So, the key is to pick a point for the center of the rotating scan that guarentees that the angle never reachese 180○.

Implementation

How do we find the point for the scanline center?

Actually, we don't care about angle – slope will do.

Select z;
 for ($i = 2$ to n)
  compute the slope of line $z - p_i$.
Sort points p_i by slope;
label points in sorted order;

Time complexity: Dominated by sort.

Pick as z the point with greatest x value (and least y value if there is a tie). See Manber Figure 8.7.

The next point is the next largest angle between $z - p_i$ and the vertical line through z. It is important to use the slope, because then our computation is a constant-time operation with no transendental functions.

z is the point with greatest x value (minimum y in case of tie)

So, time is $\Theta(n \log n)$

Convex Hull

- A **convex hull** is a polygon such that any line segment connecting two points inside the polygon is itself entirely inside the polygon.
- A **convex path** is a path of points $p_1, p_2, \cdots, p_n$ such that connecting p_1 and p_n results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - imagine placing a tight rubberband around the points.
- The point **belongs** to the hull if it is a vertex of the hull.
- **Problem**: Compute the convex hull of n points.

no notes

Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for $< n$ points, and try to add the nth point.

1. nth point is inside the hull.
   - No change.
2. nth point is outside the convex hull
   - "Stretch" hull to include the point (dropping other points).

See Manber Figure 8.9.

Subproblems (1)

Potential problems as we process points:

1. Determine if point is inside convex hull.
2. Stretch a hull.

The straightforward induction approach is inefficient. (Why?)

Our standard induction alternative: Select a special point for the nth point – some sort of min or max point.

If we always pick the point with max x, what problem is eliminated?
Stretch:

1. Find vertices to eliminate
2. Add new vertex between existing vertices.

Why? Lots of points don't affect the hull, and stretching is expensive.

Subproblem 1 can be eliminated: the max is always outside the polygon.

Subproblems (2)

Supporting line of a convex polygon is a line intersecting the polygon at exactly one vertex.

Only two supporting lines between convex hull and max point q.

These supporting lines intersect at "min" and "max" points on the (current) convex hull.

"Min" and "max" with respect to the angle formed by the supporting lines.

Sorted-Order Algorithm

set convex hull to be p_1, p_2, p_3;
for $q = 4$ to n {
  order points on hull with respect to p_q;
  Select the min and max values from ordering;
  Delete all points between min and max;
  Insert p_q between min and max;
}

Sort by x value.

Time complexity

Sort by x value: $O(n \log n)$.

For qth point:

- Compute angles: $O(q)$
- Find max and min: $O(q)$
- Delete and insert points: $O(q)$.

$$T(n) = T(n-1) + O(n) = O(n^2)$$

no notes

Gift Wrapping Concept

- Straightforward algorithm has inefficiencies.
- Alternative: Consider the whole set and build hull directly.
- Approach:
  - Find an extreme point as start point.
  - Find a supporting line.
  - Use the vertex on the supporting line as the next start point and continue around the polygon.
- Corresponding Induction Hypothesis:
  - Given a set of n points, we can find a convex path of length $k < n$ that is part of the convex hull.
- The induction step extends the PATH, not the hull.

Straightforward algorithm spends time to build convex hull with points interior to final convex hull.

Gift Wrapping Algorithm

ALGORITHM GiftWrapping(Pointset S) {
  ConvexHull P;

  $P = \emptyset$;
  Point $p =$ the point in S with largest x coordinate;
  $P = P \cup p$;
  Line $L =$ the vertical line containing p;
  while (P is not complete) do {
    Point $q =$ the point in S such that angle between line
      $-p-q-$ and L is minimal along all points;
    $P = P \cup q$;
    $L = -p-q-$;
    $p = q$;
  }
}

no notes

Gift Wrapping Analysis

Complexity:

- To add kth point, find the min angle among $n - k$ lines.
- Do this h times (for h the number of points on hull).
- Often good in average case.
- Could be bad in worst case.

$O(n^2)$. Actually, $O(hn)$ where h is the number of edges to hull.

Graham's Scan

- Approach:
  - Start with the points ordered with respect to some maximal point.
  - Process these points in order, adding them to the set of processed points and its convex hull.
  - Like straightforward algorithm, but pick better order.
- Use the Simple Polygon algorithm to order the points by angle with respect to the point with max x value.
- Process points in this order, maintaining the convex hull of points seen so far.

See Manber Figure 8.11.

Graham's Scan (cont)

Induction Hypothesis:

- Given a set of n points ordered according to algorithm Simple Polygon, we can find a convex path among the first $n-1$ points corresponding to the convex hull of the $n-1$ points.

Induction Step:

- Add the kth point to the set.
- Check the angle formed by p_k, p_{k-1}, p_{k-2}.
- If angle $< 180\circ$ with respect to inside of the polygon, then delete p_{k-1} and repeat.

no notes

Graham's Scan Algorithm

ALGORITHM GrahamsScan(Pointset P) {
  Point $p_1 =$ the point in P with largest x coordinate;
  $P =$ SimplePolygon(P, p_1); // Order points in P
  Point $q_1 = p_1$;
  Point $q_2 = p_2$;
  Point $q_3 = p_3$;
  int $m = 3$;
  for ($k = 4$ to n) {
    while (angle($-q_{m-1} - q_m-, -q_m - p_k-) \leq 180°$) do
      $m = m - 1$;
    $m = m + 1$;
    $q_m = p_k$;
  }
}

no notes

Graham's Scan Analysis

Time complexity:

- Other than Simple Polygon, all steps take $O(n)$ time.
- Thus, total cost is $O(n \log n)$.

no notes

Lower Bound for Computing Convex Hull

Theorem: Sorting is transformable to the convex hull problem in linear time.

Proof:

- Given a number x_i, convert it to point (x_i, x_i^2) in 2D.
- All such points lie on the parabla $y = x^2$.
- The convex hull of this set of points will consist of a list of the points sorted by x.

Corollary: A convex hull algorithm faster than $O(n \log n)$ would provide a sorting algorithm faster than $O(n \log n)$.

WARNING: These are the most important two slides of the semester!

“Black Box” Model

A Sorting Algorithm:

keys → points: $O(n)$

Convex Hull

CH Polygon → Sorted Keys: $O(n)$

“Black Box” Model
A Sorting Algorithm:
keys → points: $O(n)$
Convex Hull
CH Polygon → Sorted Keys: $O(n)$

This is the fundamental concept of a reduction. We will use this constantly for the rest of the semester.