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

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 \(p_1, p_2, \ldots, p_n\) and the line segments \(p_1 - p_2, p_2 - p_3, \ldots, p_{n-1} - p_n\), 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.

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.
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.

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.

Point in Polygon Analysis (2)

Two general principles for geometrical and graphical algorithms:
- 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.
- 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 $q$.

Constructing Simple Polygons

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

Approaches:
- Randomly select points.
- Use a scan line:
  - Sort points by $y$ value.
  - Connect in sorted order.
- 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.

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 segments (from a polygon intersection operation). Cascading intersections can lead to ever-increasing demand for precision in coordinate representation.

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

(1) Could easily yield an intersection.

(2) The problem is connecting point $p_1$ 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 \( p_1, \ldots, p_n \) 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°.

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.

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 \( p_1, p_2, \ldots, p_n \) 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.

Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for \( < n \) points, and try to add the \( n \)th point.

- \( n \)th point is inside the hull.
  - No change.
- \( n \)th point is outside the convex hull
  - “Stretch” hull to include the point (dropping other points).
Subproblems (1)

Potential problems as we process points:

Rather than find a hull and stretch it one point at a time, we will perform a straightforward induction approach. This is inefficient.

Our standard alternative is: 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?

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.

Sorted-Order Algorithm

set convex hull to be p1, p2, p3;
for q = 4 to n {
  order points on hull with respect to pq;
  Select the min and max values from ordering;
  Delete all points between min and max;
  Insert pq between min and max;
}

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).

\[ T(n) = T(n - 1) + O(n) = O(n^2) \]
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.

Gift Wrapping Algorithm

```
ALGORITHM GiftWrapping(Pointset S) {
    ConvexHull P;
    P = ∅;
    Point p = the point in S with largest x coordinate;
    P = P ∪ 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 ∪ q;
        L = −p − q−;
        p = q;
    }
}
```

Gift Wrapping Analysis

Complexity:
- To add \( k \)th 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.

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.

Gift Wrapping Concept

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 = ∅;
    Point p = the point in S with largest x coordinate;
    P = P ∪ 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 ∪ q;
        L = −p − q−;
        p = q;
    }
}
```

Gift Wrapping Analysis

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

Graham’s Scan

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 $k$th 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.

Graham’s Scan Algorithm

ALGORITHM GrahamScan(Pointset P) {
  Point $p_1 =$ the point in $P$ with largest $x$ coordinate;
  $P =$ SimplePolygon($p_1$); // 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($q_{m-1} - q_{m}$, $-q_{m} - p_k$) $\leq 180^\circ$) do
      $m = m - 1$;
      $m = m + 1$;
      $q_m = p_k$;
  }
}

Graham’s Scan Analysis

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

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 parabola $y = x^2$.
- 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:

\[ \text{keys} \rightarrow \text{points}: O(n) \]

Convex Hull
CH Polygon $\rightarrow$ Sorted Keys: $O(n)$

Closest Pair

- **Problem**: Given a set of $n$ points, find the pair whose separation is the least.
- **Example of a proximity problem**
  - Make sure no two components in a computer chip are too close.
- **Related problem**:
  - Find the nearest neighbor (or $k$ nearest neighbors) for every point.
- **Straightforward solution**: Check distances for all pairs.
- **Induction Hypothesis**: Can solve for $n-1$ points.
- Adding the $n$th point still requires comparing to all other points, requiring $O(n^2)$ time.

Closest Pair Algorithm

Induction Hypothesis:
- We can solve closest pair for two sets of size $n/2$ named $P_1$ and $P_2$.

Let minimal distance in $P_1$ be $d_1$, and for $P_2$ be $d_2$.
- Assume $d_1 \leq d_2$.

Only points in the strip of width $d_1$ to either side of the dividing line need to be considered.

Worst case: All points are in the strip.

Divide and Conquer Algorithm

- **Approach**: Split into two equal size sets, solve for each, and rejoin.
- **How to split**?
  - Want as much valid information as possible to result.
  - Try splitting into two disjoint parts separated by a dividing plane.
- **Then**, need only worry about points close to the dividing plane when rejoining.
- **To divide**: Sort by $x$ value and split in the middle.

Assume $n = 2^k$ points.

Note: We will actually compute smallest distance, not pair of points with smallest distance.

See Manber Figure 8.13
Closest Pair Algorithm (cont)

Observation:
- A single point can be close to only a limited number of points from the other set.
Reason: Points in the other set are at least \(d_1\) distance apart.
Sorting by \(y\) value limits the search required.

Closest Pair Algorithm Cost

\(O(n \log n)\) to sort by \(x\) coordinates.
Eliminate points outside strip: \(O(n)\).
Sort according to \(y\) coordinate: \(O(n \log n)\).
Scan points in strip, comparing against the other strip: \(O(n)\).

\[T(n) = 2T(n/2) + O(n \log n)\]
\[T(n) = O(n \log^2 n)\]

A Faster Algorithm

The bottleneck was sorting by \(y\) coordinate.

If solving the subproblem gave us a sorted set, this would be avoided.

Strengthen the induction hypothesis:
- Given a set of \(< n\) points, we know how to find the closest distance and how to output the set ordered by the points’ \(y\) coordinates.

All we need do is merge the two sorted sets – an \(O(n)\) step.

\[T(n) = 2T(n/2) + O(n)\]
\[T(n) = O(n \log n)\]

Horizontal and Vertical Segments

- Intersection Problems:
  - Detect if any intersections ...
  - Report any intersections ...
  - ... of a set of \(<\) line segments >.
- We can simplify the problem by restricting to vertical and horizontal line segments.
- Example applications:
  - Determine if wires or components of a VLSI design cross.
  - Determine if they are too close.
    - Solution: Expand by 1/2 the tolerance distance and check for intersection.

See Manber Figure 8.14
**Sweep Line Algorithms (1)**

**Problem:** Given a set of \( n \) horizontal and \( m \) vertical line segments, find all intersections between them.
- Assume no intersections between 2 vertical or 2 horizontal lines.

Straightforward algorithm: Make all \( n \times m \) comparisons.

If there are \( n \times m \) intersections, this cannot be avoided.

However, we would like to do better when there are fewer intersections.

Solution: Special order of induction will be imposed by a sweep line.

**Sweep Line Algorithms (2)**

Plane sweep or sweep line algorithms pass an imaginary line through the set of objects.

As objects are encountered, they are stored in a data structure.

When the sweep passes, they are removed.

Preprocessing Step:
- Sort all line segments by \( x \) coordinate.

**Sweep Line Algorithms (3)**

Inductive approach:
- We have already processed the first \( k - 1 \) end points when we encounter endpoint \( k \).
- Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point \( k \).

Possible approaches:
- Store vertical lines, calculate intersection for horizontal lines.
- Store horizontal lines, calculate intersection for vertical lines.

See Figure 8.17 in Manber.

**Organizing Sweep Info**

What do we need when encountering line \( L \)?
- NOT horizontal lines whose right endpoint is to the left of \( L \).
- Maintain active line segments.

What do we check for intersection?

Induction Hypothesis:
- Given a list of \( k \) sorted coordinates, we know how to report all intersections among the corresponding lines that occur to the left of \( k, x \), and to eliminate horizontal lines to the left of \( k \).
Sweep Line Tasks

Things to do:
- \((k + 1)\)th endpoint is right endpoint of horizontal line.
  - Delete horizontal line.
- \((k + 1)\)th endpoint is left endpoint of horizontal line.
  - Insert horizontal line.
- \((k + 1)\)th endpoint is vertical line.
  - Find intersections with stored horizontal lines.

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient
- Intersection
- Deletion
- 1 dimensional range query

Example solution: Balanced search tree
- Insert, delete, locate in \(\log n\) time.
- Each additional intersection calculation is of constant cost beyond first (traversal of tree).

Data Structure Requirements (2)

Time complexity:
- Sort by \(x\): \(O((m + n) \log (m + n))\).
- Each insert/delete: \(O(\log n)\).
- Total cost is \(O(n \log n)\) for horizontal lines.

Processing vertical lines includes one-dimensional range query:
- \(O(\log n + r)\) where \(r\) is the number of intersections for this line.

Thus, total time is \(O((m + n) \log (m + n) + R)\), where \(R\) is the total number of intersections.

Reductions

A reduction is a transformation of one problem to another

Purpose: To compare the relative difficulty of two problems

Example:
Sorting reals reduces to (in linear time) the problem of finding a convex hull in two dimensions
- Use CH as a way to solve sorting

We argued that there is a lower bound of \(\Omega(n \log n)\) on finding the convex hull since there is a lower bound of \(\Omega(n \log n)\) on sorting

NOT reduce CH to sorting – that just means that we can make CH as hard as sorting! Using sorting isn't necessarily the only way to solve the CH problem, perhaps there is a better way. So just knowing that sorting is ONE WAY to solve CH doesn't tell us anything about the cost of CH. On the other hand, by showing that we can use CH as a tool to solve sorting, we know that CH cannot be faster than sorting.
Reduction Notation

- We denote names of problems with all capital letters.
  - Ex: SORTING, CONVEX HULL
- What is a problem?
  - A relation consisting of ordered pairs \((I, SLN)\).
  - \(I\) comes from the set of instances (allowed inputs).
  - \(SLN\) is the solution to the problem for instance \(I\).
- Example: SORTING = \((I, SLN)\).
  - \(I\) is a finite subset of \(\mathbb{R}\).
- Prototypical instance: \(\{x_1, x_2, \ldots, x_n\}\).
  - \(SLN\) is the sequence of reals from \(I\) in sorted order.

Black Box Reduction (1)

The job of an algorithm is to take an instance \(I\) and return a solution \(SLN\), or to report that there is no solution.

A reduction from problem \(A(I, SLN)\) to problem \(B(I', SLN')\) requires two transformations (functions) \(T, T'\):

\[- T: I \Rightarrow I' \Rightarrow I'' \]
  - Maps instances of the first problem to instances of the second.
\[- T': SLN' \Rightarrow SLN \Rightarrow SLN'' \]
  - Maps solutions of the second problem to solutions of the first.

Black Box Reduction (2)

Black box idea:

1. Start with an instance \(I\) of problem \(A\).
2. Transform to an instance \(I' = T(I)\), an instance of problem \(B\).
3. Use a “black box” algorithm for \(B\) as a subroutine to find a solution \(SLN'\) for \(B\).
4. Transform to a solution \(SLN = T'(SLN')\), a solution to the original instance \(I\) for problem \(A\).
More Notation

If \((I, \text{SLN})\) reduces to \((I', \text{SLN}')\), write:
\[(I, \text{SLN}) \leq (I', \text{SLN}').\]

This notation suggests that \((I, \text{SLN})\) is no harder than \((I', \text{SLN}')\).

Examples:
- \(\text{SORTING} \leq \text{CONVEX HULL}\)

The time complexity of \(T\) and \(T'\) is important to the time complexity of the black box algorithm for \((I, \text{SLN})\).

If combined time complexity is \(O(g(n))\), write:
\[(I, \text{SLN}) \leq O(g(n)) (I', \text{SLN}').\]

Reduction Example

\(\text{SORTING} = (I, \text{SLN})\)
\(\text{CONVEX HULL} = (I', \text{SLN}').\)

- \(I = \{x_1, x_2, \ldots, x_n\}\).
- \(T(I) = I' = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}\).
- Solve \(\text{CONVEX HULL}\) for \(I'\) to give solution \(\text{SLN}'\)
  \[= \{(x_1^2, x_1^2), (x_2^2, x_2^2), \ldots, (x_n^2, x_n^2)\} \]
- \(T'\) finds a solution to \(I\) from \(\text{SLN}'\) as follows:
  - Find \((x_{i[k]}, x_{i[k]}^2)\) such that \(x_{i[k]}\) is minimum.
  - \(Y = x_{i[k]}, x_{i[k] + 1}, \ldots, x_{i[k] + 1}, \ldots, x_{i[k] + 1}\).
  - For a reduction to be useful, \(T\) and \(T'\) must be functions that can be computed by algorithms.
  - An algorithm for the second problem gives an algorithm for the first problem by steps 2 \(-\) 4.

Notation Warning

Example: \(\text{SORTING} \leq O(n) \text{ CONVEX HULL}\).

WARNING: \(\leq\) is NOT a partial order because it is NOT antisymmetric.

\(\text{SORTING} \leq O(n) \text{ CONVEX HULL}\).

\(\text{CONVEX HULL} \leq O(n) \text{ SORTING}\).

But, \(\text{SORTING} \neq \text{CONVEX HULL}\).

Bounds Theorems

**Lower Bound Theorem:** If \(P_1 \leq O(g(n)) P_2\), there is a lower bound of \(\Omega(h(n))\) on the time complexity of \(P_2\), and \(g(n) = o(h(n))\), then there is a lower bound of \(\Omega(h(n))\) on \(P_2\).

Example:
- \(\text{SORTING} \leq O(n) \text{ CONVEX HULL}\).
- \(g(n) = n, h(n) = n\log n, g(n) = o(h(n))\).
- Theorem gives \(\Omega(n \log n)\) lower bound on \(\text{CONVEX HULL}\).

**Upper Bound Theorem:** If \(P_2\) has time complexity \(O(h(n))\) and \(P_1 \leq O(g(n)) P_2\), then \(P_1\) has time complexity \(O(g(n) + h(n))\).

Notice \(o\), not \(O\). So, given good transformations, both problems take at least \(\Omega(P_1)\) and at most \(O(P_2)\).