11. Computational Geometry and Convex Hulls

CPSC 535 ~ Spring 2019

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

November 18, 2019

Big Idea: Output Sensitive Algorithm

- **input sensitive**: time efficiency is a function of the input e.g. size n, # edges m
- **output sensitive**: efficiency is also a function of the *output* size e.g. # items returned
- most relevant when the size of the output could be the bottleneck

Computational Geometry

computational X: interdisciplinary study of computer science with X
(computational sociology/epidemiology/physics/finance/etc.)

computational geometry (CG): algorithms, data structures, asymptotic analysis, of geometric objects: points, lines, circles, triangle meshes, etc.

Applications

- computer graphics, user interfaces
- GIS, geographic databases
- scene reconstruction (e.g. LIDAR)
- business operations research (e.g. linear programming, aircraft control)
- manufacturing (e.g. feasibility of assembly, castings)

Putting the Geo in CG

Some general algorithms can actually solve geometric problems efficiently, without any awareness of the geometry.

bounding box problem
input: set of 2D points $P = \{p_1, p_2, \ldots, p_n\}$
output: points $tl = (x_l, y_t)$ and $rb = (x_r, y_b)$ such that the rectangle with top-left corner tl and bottom-right corner rb contains P

Naïve, optimal algorithm: x_l, y_t, x_r, y_b = min x, min y, max x, max y respectively; $\Theta(n)$

Computational geometers are most interested when geometric properties matter.

Line Segment Predicates

We can use arithmetic to answer any of the following predicates (questions) about points p_0, p_1, p_2, p_3 in $\Theta(1)$ time:

1. Is line segment $\overline{p_0 p_1}$ clockwise from $\overline{p_0 p_2}$ around the common endpoint p_0?
2. If we follow $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we turn right or left?
3. Do line segments $\overline{p_0 p_1}$ and $\overline{p_2 p_3}$ intersect?

$\Longrightarrow$ We may use any of these in pseudocode.

Degeneracy and Non-Degeneracy Assumptions

degenerate object: has the proper shape/type, but the values are a special case that betrays the spirit of the definition

Example: triangle $\equiv$ three points (p_1, p_2, p_3)
degenerate triangle: $p_1 = p_2 = p_3$; or p_1, p_2, p_3 colinear; etc.

Non-degeneracy assumption:

- constraint that input to a CG algorithm is not degenerate in specific ways
- simplifies algorithm design
- assume that in practice, some combination of
  - degeneracies do not occur
  - input can be preprocessed to remove degeneracies
  - implementer can modify algorithm to handle degeneracies

Sweep Algorithms

A pattern in CG algorithms:

- *line sweep:* envision a line "sweeping" through the input
- e.g. a vertical line sweeping left-to-right
- helps us visualize a 2D situation as a 1D situation that changes over time
- like duality, doesn't actually change the problem, but might help us problem-solve
- generalizes to higher dimensions e.g. plane sweep in 3D, hyperplane sweep in any dimension

Convex Hulls

convex hull problem
input: set of $n \geq 3$ points Q
output: $CH(Q)$, the subset of Q that is the set of vertices on the convex hull of Q

Convex hull $\equiv$ boundary of convex polygon enclosing all of Q

Applications

- object intersection in raytracing, video games, GUIs
- drawing implicit regions in GIS
- finding farthest points
- component of other algorithms

Approaches to Convex Hulls

Like the sorting problem, many algorithm patterns work for convex hulls, and there is a rich literature of competitive algorithms.

- Greedy pattern: line-sweep, update hull as we go
- Divide-and-conquer: divide Q in half, compute convex hulls for each half, merge two convex hulls into one
- Iterative improvement: start with a superset of $CH(Q)$; refine by repeatedly eliminating a constant fraction of the points until only $CH(Q)$ remains

Baseline Algorithm

Observe

- any two input points define a line ℓ
- when those points are both in $CH(Q)$, remaining $n-2$ points are all on the same side of ℓ *(geometric property)*
- $\Longrightarrow$ for each pair of input points p, q, see whether all other points are on the same side of ℓ
- if so include p, q in $CH(Q)$

Baseline Pseudocode

```
function NAIVE-CONVEX-HULL(Q)
    H = ∅
    for distinct points p, q ∈ Q do
        form line ℓ intersecting p and q
        k = # points above ℓ
        if k = (n − 2) or k = 0 then
            H = H ∪ {p, q}
        end if
    end for
    return H
end function
```

Analysis: $\Theta(n^3)$ time

Graham Scan Idea

- greedy pattern, reduction-to-sorting
- Heuristic: when touring the hull in counter-clockwise order, we **only make left turns**
- right turn = exiting a concavity; middle point not in hull
- ∴ sweep counter-clockwise, keep points that participate in left turns, drop points in the middle of right turns
- alternative kind of line sweep: rotating the line (not left-to-right)

Graham Scan Greedy Heuristic

- $p_1, \ldots, p_m = Q$ sorted into counter-clockwise order, eliminating ties
- stack S of points; contains hull of points visited *already*
- base case: push first 3 points onto S
  - for any three points p, q, r forming a non-degenerate triangle, $CH(\{p, q, r\}) = \{p, q, r\}$
- inductive case:
  - examine next input point p_i, top of stack t, next-lowest stack point r
  - if $\angle rtp_i$ is not a left turn $\implies$ t not on hull
- Note: need stack data structure w/ accessor to top **two** elements

Graham Scan Pseudocode

```
function GRAHAM-SCAN(Q)                          ▷ guaranteed |Q| ≥ 3
    p0 = lowest point in Q (break ties by choosing leftmost point)
    p1 ... pm = sort Q − {p0} into counter-clockwise order, by polar
  angle with p0; break ties by keeping only the point farthest from p0
    S = new stack
    S.PUSH(p0)
    S.PUSH(p1)
    S.PUSH(p2)
    for i from 3 through m do
        while ∠pi, S.TOP, S.BELOWTOP is non-left turn do
            S.POP()
        end while
        S.PUSH(pi)
    end for
    return S
end function
```

Graham Scan Analysis

- find p_0: $\Theta(n)$
- sort: $\Theta(n \log n)$
- eliminate tied points: $\Theta(n)$
- each stack operation is $\Theta(1)$
- **for** loop repeats $m < n$ times
- turn angle test, stack operations are $\Theta(1)$
- $\Rightarrow \Theta(n \log n)$ time
- dominating term is sort, organizing data structure is arrayed stack $\Rightarrow$ good constant factors

Jarvis March

Alternative greedy heuristic: moving around the hull counter-clockwise, each step from one vertex to the next is *the input point whose angle is shallowest* ("gift wrapping")

Jarvis march

1. Find the lowest and highest points in Q.
2. (right chain) Starting from the lowest point, and until we reach the highest point:
   2.1 Linear search Q for the next point, minimizing the angle between the two points.
   2.2 Add the first point to $CH(Q)$ and move to the second point.
3. (left chain) Starting from the highest point, repeat this process until we reach the lowest point.
4. Return $CH(Q)$

Jarvis March Analysis

Preprocessing to find highest/lowest: $\Theta(n)$

Each iteration of the left/right-chain loops identifies one hull point $\implies$ in total they iterate h times, where $h \equiv$ number of points on the hull.

linear search inside the loops takes $\Theta(n)$ time.

$\therefore \Theta(nh)$ total time.

Faster than Graham scan's $\Theta(n \log n)$ when $h \in o(\log n)$.

Optimal output-sensitive: Chan's algorithm, $\Theta(n \log h)$.

Summary of Convex Hull Algorithms

Algorithm	Time	Main Idea
Graham Scan	$\Theta(n \log n)$	sort, skip right turns
Jarvis March	$\Theta(nh)$	gift-wrapping
Chan's algorithm	$\Theta(n \log h)$	divide w/ Graham, merge w/ Jarvis