Computational Geometry Homework 3

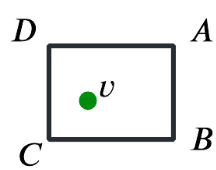
Ian Sinclair
COMP 3705-1

March 30, 2022
Dr. Mario Lopez

Problem 1

Implement a full efficient solution to the orthogonal range counting problem for a set S of n points. Test your solution with a driver that allows you to either (a) enter the points interactively, or (b) read them from a file, and query with arbitrary upright rectangles $R(a, b, c, d)$.

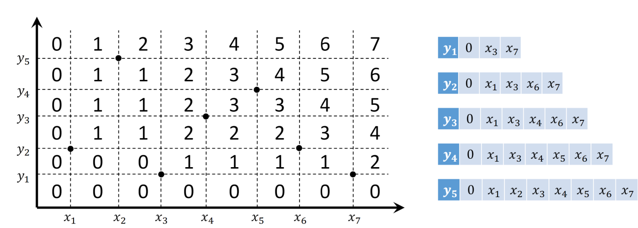
For a subset S_i of s is bounded by R, every point $v \in S_i$ must strictly bounded in 4 directions by a permutation of $\{a, b, c, d\}$. As a result this can become a domination problem. Without loss of generality, let a permutation of $\{a, b, c, d\}$ be defined in the following configuration.



Then for any point $v \in S_i$ v must be dominated by A. Such that, $v \in D(A)$. Where $D(A)$ is the set of points in S dominated by A.
Additionally, it must be true that, $v \notin D(B) \cup D(C) \cup D(D)$.
And so v is in the query rectangle if,

$$v \in D(A) \backslash \{D(B) \cup D(C) \cup D(D)\}$$

From here note a locus method to preprocess the points in S, by ordering them with respect to increasing y. Then increasing x.



Then, determining if a query point u dominates any point in S, is the same as finding its location in loci, where u dominates everything in its y column with a lower x value.

```
"""
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

              HOMEWORK 3

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

```

```
-----------------------------------
    Orthogonal Range Counting
-----------------------------------
"""

def locus_preprocess( points : list ) :
    y_sort_points = sorted(points, key=lambda v: v.y)
    loci = {}

    for v in y_sort_points :
        #loci[v.y] = [ point(0,0) ]
        loci[v.y] = []
        loci[v.y] += sorted(y_sort_points[0:y_sort_points.index(v)+1], key=lambda q: q.x)


    return loci

def dominates( u : point, locus : dict ) :
    keys = locus.keys()
    q_y = 0

    dominated = []

    for i in keys :
        if i <= u.y:
            q_y = i
        else :
            break

    if q_y == 0 :
        return []

    for v in locus[q_y] :
        if v.x <= u.x:
            dominated += [v]

    return dominated


def orthogonal_range_counting( R : rectangle, Locus: dict) :
    y_max = max([R.a.y, R.b.y, R.c.y, R.d.y])
    x_max = max([R.a.x, R.b.x, R.c.x, R.d.x])

    y_min = min([R.a.y, R.b.y, R.c.y, R.d.y])
    x_min = min([R.a.x, R.b.x, R.c.x, R.d.x])


    A = point(x_max,y_max)
    B = point(x_max,y_min)
    C = point(x_min,y_min)
    D = point(x_min, y_max)

    dom_A = dominates(A , Locus)
    dom_B = dominates(B , Locus)
    dom_C = dominates(C , Locus)
    dom_D = dominates(D , Locus)

    for v in dom_C :
        dom_B.remove(v)

    for v in dom_B + dom_D :
        dom_A.remove(v)

    return dom_A
```

Problem 2

Let $P =< p_0, p_1, ..., p_{n-1} >$ with $p_i = (x_i, y_i)$ be a simple polygon stored as a list of vertices in CCW order around ∂P, Vertices are stored using rational coordinates. A chord of P is a segment interior to P whose endpoints belong to ∂P. For example, all diagonals are chords.
Describe a data structure to determine, as fast as possible, the area of the two subpolygons of P induced by a chord C.

i) When P is convex.

ii) When P is an arbitrary simple polygon.

Your data structure should require $O(n)$ space. What is the time complexity of your algorithm?

When P is convex.

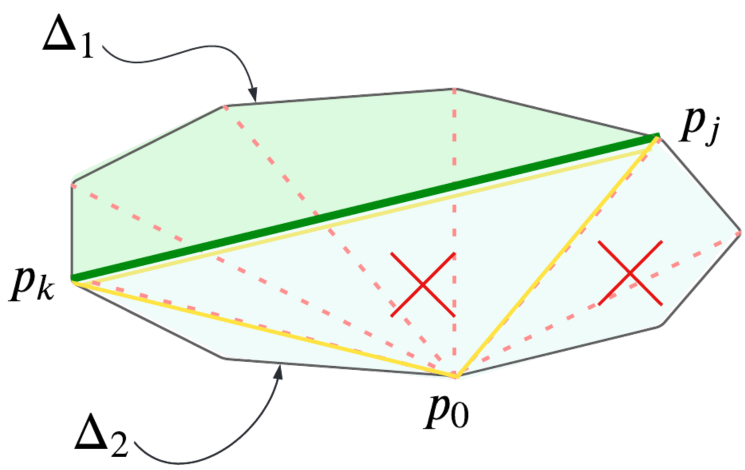
Let $P =< p_0, p_1, ..., p_{n-1} >$, be any convex polygon. Then select p_0 as a root and calculate the area of the triangle created by adjoining p_0 to each other not neighboring vertex through a chord.
And so without loss of generality, any one area is found by,

$$\mathcal{A}(p_0, p_i, p_{i-1}) = \frac{1}{2}[(p_1 - p_0) \times (p_{i-1} - p_0)].$$

Now, incrementally store the sum of areas leading to vertex p_i in a hash table.

$$h(i) = \sum_{k \leq i} \mathcal{A}(p_0, p_k, p_{k-1})$$

And so, the table at index i is the area of the subpolygon $p_i =< p_0, p_1, ..., p_i >$.
Now, if a query chord is a diagonal to p_0, with end points (p_0, p_i) the solution is trivial, the first area is the exact sum in the hash table, $h(i)$ then the second is the total area subtracted by the value in the hash table at $h(n) - h(i)$. Which has constant time. If the chord does not have p_0 as an end-point, has end-points, (p_j, p_k) for $j, k \neq 0$. Then without loss of generality take $j \leq k$, then the area of one-polygon is defined by, $h(k) - h(j) - \mathcal{A}(p_0, p_j, p_k)$. And then of course the area of the second subpolygon is $h(n) - h(k) + h(j) + \mathcal{A}(p_0, p_j, p_k)$.



Here, $h(k)$ is the area of the subpolygon, $< p_0, p_1, ..., p_k >$. And similarly $h(j)$ is the area of the subpolygon $< p_0, p_1, ..., p_j >$. Then the area, Δ_1 is,

$$\Delta_1 = h(k) - h(j) - \mathcal{A}(p_0, p_k, p_j).$$

And,

$$\Delta_2 = h(n) - h(k) + h(j) + \mathcal{A}(p_0, p_k, p_j).$$

Where $h(n)$ is the area of the polygon P.

Solution

And so because the algorithm is a hash table,

Space: $O(n)$, Time: $O(c)$, Preprocessing: $O(n)$

When P is an arbitrary simple polygon.

Let $P =< p_0, p_1, ..., p_{n-1} >$ by any simple polygon. Then the above method can be generalized to also apply. This is because the area of sub-polygons are in-variant to negative areas in subsections. Or in general, for chord with endpoints, (p_j, p_k), with out loss of generality let $j \leq k$, then the area of the sub-polygon $p_k = (p_0, p_1, ..., p_k)$, is given by the hash table h, $h(k)$, regardless of negative area triangles in the polygon p_k (or has at least one reflex vertex). And so the algorithm holds for any simple polygon.

Solution

$$\Delta_1 = h(k) - h(j) - \mathcal{A}(p_0, p_k, p_j)$$
$$\Delta_2 = h(n) - \Delta_1 = h(n) - h(k) + h(j) + \mathcal{A}(p_0, p_k, p_j)$$

And so because the algorithm is a direct look up hash table,

Space: $O(n)$, Time: $O(c)$, Preprocessing: $O(n)$