Introduction

Computational geometry looks at computational questions that have geometric inputs, outputs, &/or queries.

We hope to learn how to
1) Translate intuition to rigorous arguments
2) Abstract thinking
3) Understanding algorithms beyond 1 dimension (e.g. trees, sorting, basic graph search)

Euclidean Geometry (4th century BCE)

Euclid's Elements was the standard geometry textbook for 2000 years. It axiomatically defined 2D & 3D geometry.

Here are Euclid's axioms
1) You can draw a straight line b/w 2 points
2) You can extend any line.
3) You can draw a circle w/ a point & line/radius.
4) All right angles are equal.
*5) Parallel lines never meet.

(5) is called the parallel postulate & isn't obviously true. In fact on the surface of a sphere it doesn't hold.

Points & Linear Algebra/Vectors

A point is a point in space. We identify them w/ coordinates.

Coordinates can be identified w/ vectors as long as you have some origin/zero.

However, points/coordinates are NOT the same thing as vectors. For example adding two points don't make sense. We'll go thru ones that make sense.

- vector + vector = vector
- vector + point = point
- point + point = undefined
- scalar · vector = vector
- scalar · point = undefined
- vector - vector = point
- vector - point = undefined
- point - vector = point
- point - point = vector (arrow from right to left)

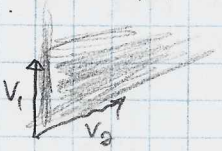
Notation:
To make writing ranges easier, $[n] = \{0, \ldots, n-1\}$.
↳ like range(n) in python.

Recall a **linear combination** is a scaled sum of vectors. That is given
$v_0, \ldots, v_{n-1} \in \mathbb{R}^k$
$\alpha_0, \ldots, \alpha_{n-1} \in \mathbb{R}$
$u = \sum_{i \in [n]} \alpha_i v_i$
We say u is a linear combination of $\{v_0, \ldots, v_{n-1}\}$.

Def:
If we restrict $\alpha_i \geq 0 \ \forall i \in [n]$, we get a **non-negative (linear) combination**

Def:
The space of all possible non-negative combinations of vectors $\{v_0, \ldots, v_{n-1}\}$ is called a **cone**.

For example, given a 2D example



Def:
An **affine combination** is a linear combination where $\sum \alpha_i = 1$.

Def:
A **convex combination** is a linear combination which is non-negative & affine.

Def:
The **determinant** is a function $\mathbb{R}^{n \times n} \to \mathbb{R}$. It can be viewed in many different ways.

You can view it by how a matrix scales the unit cube (or similar in space. When $\det(A) = 0$, the matrix / linear transformation is degenerate.

You can view it as a way to geometrically extend comparisons.

Take some $a, b \in \mathbb{R}$.
$a \geq b \iff a - b \geq 0 \iff \det\begin{bmatrix} a & b \\ 1 & 1 \end{bmatrix} \geq 0$

Now consider some vectors $a_1, \ldots, a_n \in \mathbb{R}^n$
$\det\begin{bmatrix} a_1 & \cdots & a_n \\ 1 & & 1 \end{bmatrix} \geq 0$ is similar to comparison

W/ this we can ask whether a point is above or below a line. Or extending it further is a point above or below a hyperplane?

We form a hyperplane w/ $a_1, \ldots a_{n-1}$ & make an the point

Thm:
Given $a, b, c \in \mathbb{R}^n$

$$\det\begin{bmatrix} b-a & c-a \\ \vdots & \vdots \end{bmatrix} = \det\begin{bmatrix} a & b & c \\ 1 & 1 & 1 \end{bmatrix}$$

Note: The permanent is like the determinant but w/ no alternating + & −

We can formulate the determinant using permutations. using

Let M_{ij} be the (i,j) entry of $M \in \mathbb{R}^{n \times n}$.
Let $[n] = \{0, \dots, n-1\}$ & σ be a permutation $[n] \to [n]$.
Then

$$\det(M) = \sum_{\sigma} \left((-1)^{\operatorname{sgn}(\sigma)} \prod_{i \in [n]} M_{i\sigma(i)}\right)$$

Row reduction is just factoring out matrixes

This formulation (& the recursive cofactor expansion) takes $O(n!)$ b/c there are $n!$ possible permutations.

But in practice, we can calculate the determinant in $O(n^3)$ time. How?

In practice we calculate the determinant by row-reducing to a triangular matrix, which we can do in $O(n^3)$ time. Then we just take the product of the diagonal.

This works b/c row reduction does not change the determinant (assuming you don't swap rows). You can prove this b/c $\det(AB) = \det(A)\det(B)$ $\forall A, B \in \mathbb{R}^{n \times n}$ & $\det(e) = 1$ for all (necessary) row operations represented as matrixes e.

Note: Oh yeah, every elementary operation on a matrix can be viewed as a multiplication by an elementary matrix e.

Predicate

We want to boil as many of our properties/problems/queries into simple calculations. To do this, we use linear predicates.

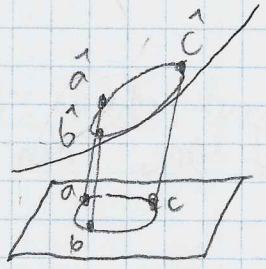
Def:
A linear predicate is the sign of of our primitive operations (positive, negative, or zero).

Incircle Tests

Given 3 points that define a triangle, find the circumcenter of that triangle. That is, find the circle that exactly touch those 3 points/vertices.

In particular, given points a, b, c that define a circumcenter w/ center v & radius r

To solve this, we will lift our points off the plane into 3D.



$$\hat{a} = \begin{bmatrix} a \\ \|a\|^2 \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} b \\ \|b\|^2 \end{bmatrix}, \quad \hat{c} = \begin{bmatrix} c \\ \|c\|^2 \end{bmatrix}$$

This is called a parabolic lifting.

The key w/ this parabolic lifting is it "linearizes" problems about the circle.

Let $v' = \begin{bmatrix} -2v \\ 1 \end{bmatrix}$ be the normal vector to the plane it thru a, b, c.

Since v' normal to the plane we know $v'^T(b-a) = 0$ (& also $v'^T(c-a) = 0$).

$$v'^T(b-a) = 0 \Rightarrow v'^T\hat{a} = v'^T\hat{b}$$
$$\Rightarrow -2v^Ta + \|a\|^2 = -2v^Tb + \|b\|^2$$
$$\Rightarrow \|v\|^2 - 2v^Ta + \|a\|^2 = \|v\|^2 - 2v^Tb + \|b\|^2$$
$$\Rightarrow \|v-a\|^2 = \|v-b\|^2$$

Similarly $\|v-a\|^2 = \|v-c\|^2$, so v is the circumcenter of (a, b, c).

We claim that for a given point p
p is above $H \Leftrightarrow p$ is outside the circumcircle(a, b, c).

This is true b/c $\|p\|^2 > 2v^T(p-a) + \|a\|^2$ iff
$\|p-v\|^2 > \|a-v\|^2 =$ circumradius(a, b, c)

Similarly iff p on plane then p on circle. If p below plane then p in circle.

We have thus reduced the incircle test to a planeside test (done w/ linear predicates)

Our final "code" needs to normalize the order of a, b, c.
This is b/c

$$\det\begin{bmatrix} \hat{a} & \hat{b} & \hat{c} & \hat{q} \\ 1 & 1 & 1 & 1 \end{bmatrix} = -\det\begin{bmatrix} \hat{b} & \hat{a} & \hat{c} & \hat{q} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

where a, b, c describe the circle & q is our query point.
We don't want our results to depend on the order of a, b, c.
To fix this, we do (with

$$\text{incircle}(a, b, c, q) = \frac{\text{orient}\begin{bmatrix} a & b & c & q \\ 1 & 1 & 1 & 1 \end{bmatrix}}{\text{orient}(a, b, c)}$$

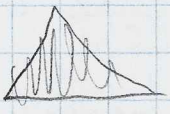
where orient = sign ∘ det.

Convexity

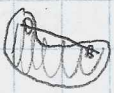
Def:
A set $X \subseteq \mathbb{R}^d$ is <u>convex</u> if for all $a, b \in X$, the line segment $\overline{ab} \subseteq X$.

Recall formally
$$\overline{ab} = \{ (1-t)a + tb \mid t \in [0,1] \}$$

<u>Examples:</u>
$\{a\}$ where $a \in \mathbb{R}^d$ is convex.

A triangle is convex. 

An ellipse is convex 

A bean is not convex 

A torus is not convex 

Two balls are not convex 

<u>Thm:</u>
Given a convex set X w/ points $a, b, c \in X$. Every convex combination of a, b, c are themselves convex subsets of X.

In other words, convex sets are closed under convex combinations.

Recall:
A convex combination is a linear combination that is both non-negative & affine.

Non-negative combinations have only positive scalars. Affine scalars have their scalars sum to 1.

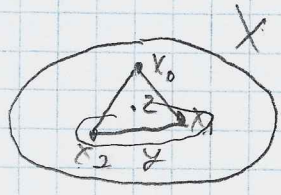
Note that we will be dealing w/ finite combinations.

Note: Sometimes our orient() function is called ccw() (for counter-clockwise test)

Let P be a set of points. If $x \in \text{conv}(P)$ is a convex combination of the points, then
$x = \sum_{i \in [n]} \alpha_i p_i$ where $p_i \in P$

& $\sum_{i \in [n]} \alpha_i = 1$ (affine) & $\alpha_i > 0 \ \forall i \in [n]$ (non-negative)

Pf:
What we want to do is given some convex set X w/ points $\{x_i\}_{i \in [n]} \in X$, we want to show
$z = \sum \alpha_i x_i$ where $\sum \alpha_i = 1$ & $\alpha_i > 0$ is in X
That is, we show that X is closed under convex combination.



Let $y = \frac{\alpha_0 x_0 + \dots + \alpha_{n-2} x_{n-2}}{1 - \alpha_{n-1}}$ ← rescale so its affine

Recall $z = \alpha_0 x_0 + \dots + \alpha_{n-1} x_{n-1}$.

We assume $y \in X$. z is a convex combination of x_{n-1} & y, so it is part of the line segment $\overline{x_{n-1} y}$. By definition convex sets are closed under line segments, so $\overline{x_{n-1} y} \subseteq X$.

We know $y \in X$ by induction b/c we can keep recursing until we get to the case case which is $x_0 \in X$.

Therefore all convex sets X are closed under convex combination.

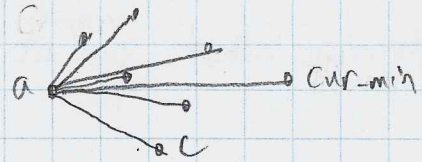
The Convex Hull Problem

Def:
Let P be a set of points. The convex hull is the minimum (convex) polygon A such that $P \subseteq A$.

Minimum polygon is not defined for us but intuitively it's like the points are nails & you're wrapping them in string.

The convex hull problem has the following specification
Input: n points in $\mathbb{R}^2$
Output: The convex hull for the points

Let's tackle a simpler problem, finding the lower hull where the points are already sorted.
Problem:



Here we're just trying to find the first edge.
Notice that the correct edge is the steepest down.
In fact what we want to do is find the point w/ minimum slope. We can get around calculating all their slopes by

```
a, cur_min, *_ = P
for c in P[1:]:
    if orient(a, cur_min, c):
        cur_min = c
return cur_min
```

This algorithm is $O(n)$ for a single edge but $O(n^2)$ for all edges.

We can actually do better w/ $O(n)$ in total (assuming points are sorted). It's called the Graham Scan.

Algorithm: Graham Scan Presorted
The idea here is to keep a stack of points.

You start on the leftmost point & walk to the right. Every time you take a step if you turned right, pop points until you are no longer turning right from the point on the top of the stack. Then push your current point onto the stack & take another step.

You know if a point is turning right or left using orient() (or CCW()).

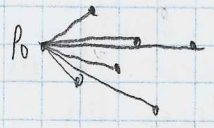
See the pseudocode on the next page.

```
stack = []
for p in points:
    while len(stack) > 1 and orient(stack[-2], stack[-1], p) <= 0:
        stack.pop()
    stack.push(p)
return stack  # points for convex hull
```

Def:
A <u>half-space</u> is a space split in half by a hyperplane.

We can do the Graham Scan w/ unsorted points. What you do is use orient() (or ccw()) to sort the points.

<u>Algorithm</u>: Graham Scan Unsorted
Given some set of points P & pick a special point p_0



Then for any points $a, b \in P \setminus \{p_0\}$ we say $a \leq b$ if $ccw(p_0, a, b)$.
As visualized, you start w/ the point w/ the most negative angle.

How do you pick p_0? One way is to make a point extremely far away from the set P & then sort & start w/ the min point.

You can set a point an infinite distance away in our projection to $z=1$ by making a point a $z=0$.

Note that the Graham scan is an iterative algorithm & at every step you have a convex hull for the points seen.

Much like Graham scan is like insertion sort, there's mergehull for the same problem

<u>Algorithm</u>: Mergehull
We're going to assume we know how to efficiently split a set of points in two & how to make a convex hull in the base case.

Therefore we are given two disjoint convex hulls L & R that we want to merge into 1 convex hull.

(This is linear b/c each point is considered at most once)

The main idea is to pick 2 random points $l \in L$ & $r \in R$. & march them until we reach the correct location. The line is $\overline{lr}$
First choose counterclockwise on hull L & clockwise on hull R.
Starting w/ hull L, check if the counterclockwise point on hull L is a right turn the line drawn. If it is, the march point L to that.
Do that until you're stuck. Then switch to marching the point on R.
Do that until you're stuck. Keep switching until both are stuck.
The final line segment is the correct edge.

Do that moving L & R in the opposite direction & checking for points on the left. Then you get your other line.

This doesn't always work?

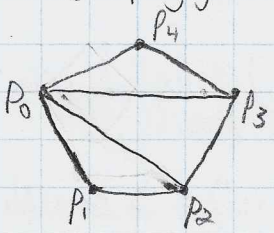
We want to prove a lower bound on convex hull problems. We do this via a reduction to comparison based sorting which if you recall has a $\Omega(n \log(n))$ lower bound.

Polygons

Lets first talk about determining if a polygon contains a point. A naive algorithm for determining if a convex polygon P contains some arbitrary point x is checking if every edge has the same orientation wrt x. So in pseudo-Python

```
wanted_orientation = orient(p0, p1, x)
return all(
    orient(pi, pi+1, x) == wanted_orientation
    for i in range(1, n)
)
```

We can improve this w/ a binary search stlye algorithm. The idea is to pick some arbitrary point p_0 in polyon P. We then split the polygon into triangles w/ edges $\overrightarrow{p_0 p_i}$ $\forall i \in [1, \ldots, n-1]$.



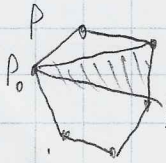
Essentially, the solution is

```
def contains(P, x):
    if orient(P0, P1/2, x):
        return contains(P[1/2:], x)
    else:
        return contains(P[:1/2+1], x)
```

Here the time is $O(\log N)$.

Area

In the membership test work, we split up the polygon into triangles. We can use this to measure the area of the polygon.



$\text{Area}(p_0 p_i p_{i+1}) = \frac{1}{2} \det[(p_i - p_0) \; (p_{i+1} - p_0)]$

Think about the parallelogram of $\overrightarrow{p_0 p_i}$ & $\overrightarrow{p_0 p_{i+1}}$

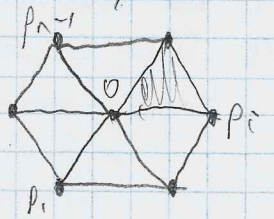
Thus we know that the area of P is

$$\text{Area}(P) = \sum_{i=1}^{n-2} \tfrac{1}{2} \det[(p_i - p_0) \quad (p_{i+1} - p_0)]$$

It doesn't seem possible to compute this in linear time...

We can avoid the subtracting tho by picking the origin point O.

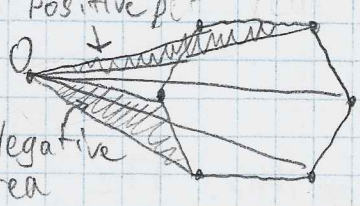
If $O \in P$, then



$$\text{Area}(P) = \sum_{i=0}^{n-1} \tfrac{1}{2} \det[p_i \quad p_{i+1}]$$

(implicitly subtract by O)

If $O \notin P$, then this still works b/c you just get negative area!



$$\text{Area}(P) = \sum_{i=0}^{n-1} \tfrac{1}{2} \det[p_i \quad p_{i+1}]$$

(implicitly subtract by O)

Now that we can calculate area, we can define a new problem.

Problem: The Most Balanced Chord

Given a convex polygon P, find i, j s.t. $\overline{p_i p_j}$ divides P into polygons whose area is as close to $\frac{1}{2}\text{Area}(P)$ as possible.

Equivalently, maximize $\text{Area}(P[i:j+1])$ s.t. $\text{Area}(P[i:j+1]) \leq \frac{1}{2}\text{Area}(P)$.

Solution 1: Brute Force

Just try every possible chord.

$\binom{n}{2} = n$ chords

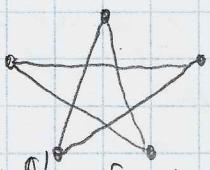
$O(n)$ time to compute areas?

Future Idea: Can we improve this?

Simple Polygons

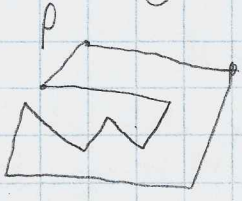
A polygon is simple iff it has no self-intersections. (i.e. edges that cross).

Simple



Non-Simple

Let's figure out the area of a simple polygon like



We'll show that our old algorithm actually works

$$\text{Area}(P) = \sum_{i=[1,7]} \tfrac{1}{2} \det[p_i \; p_{i+1}]$$

To do this, we need to show that every point inside is counted once & outside is never counted.

To do this, we need the Jordan Curve Theorem.

Thm:

The [polygonal] Jordan curve theorem states that every simple polygon divides the plane into two pieces, an inside & outside.

See next page.

Further, let $p \in \mathbb{R}^2$ be a point & $v \in \mathbb{R}^2$ be a vector.
Consider a ray $r = \{p + tv \mid t \geq 0\}$.

The point p is inside polygon P iff there are an odd number of crossings b/w r & the edges of P.

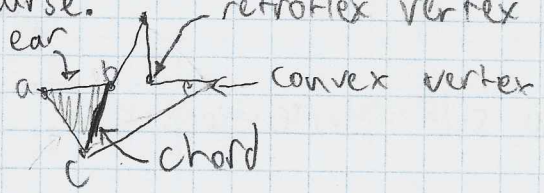
The point p is outside iff there are an even number.

Simple Polygons

We want to develop an efficient algorithm for classifying a polygon as simple (non-intersecting) or non-simple.

Let's first try to solve the problem of triangulating a polygon.

A simple idea is to find an ear in the polygon, cut it off, & then recurse.



To find an ear, we need to first find a convex vertex (linear time). Then we draw the chord & ensure no vertexes are contained in the triangle (linear time). We then recurse n times giving us a $O(n^3)$ algorithm.

Note: This assumes an ear exists

We can improve this by just finding a chord.

Let's now show that such a chord exists & how to find it.

Consider some triangle drawn w/ a convex vertex (like we did last time). Draw a vertical line & sweep it from the left side of the triangle to the right. The first vertex we hit w/ the line that is in the triangle, draw a line b/w the leftmost vertex of the triangle & the vertex we hit.

This line is guaranteed to be a chord. If we hit no vertexes inside of the triangle, then the line we drew for the triangle is a chord.

We have thus shown that a chord exists.

Let's show our earlier assumption about ears holds.

thm:
Every triangulation of a simple polygon (w/ at least 4 vertices) & has at least two ears.

pf:
Let's do an inductive proof on the # of vertices, v.

If there are 4 vertices, then clearly there are two ears.

Not clear but easy to believe

Now suppose $v > 4$. Cut along the chord (such a chord exists by earlier work)

Each of the two parts either is a triangle or has 2 ears. Gluing together two parts loses ≤ 2 ears. The result is

$\underbrace{2}_{\text{gained from parts}} + 2 - 2 = 2$

Now let's create a $O(n \log n)$ algorithm for triangulating a polygon

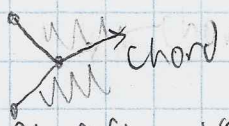
Algorithm: Triangulate Polygon in $O(n \log n)$

the technique is to keep track of line segments as you sweep a line over the edges/vertexes of the polygon.

As we sweep a line to the right, we

- insert (left end)
- delete (right end)
- above/below

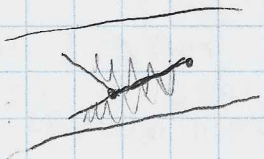
When we hit a left-reflex vertex, we want to find a chord that goes to the right.



left reflex vertex

We split into cases to draw the chord

We knew what the edges were above/below that left reflex-vertex. If some new vertex appears that has the same edges above/below it, we draw a chord to it



If one of those edges above/below that left reflex vertex stop before another vertex appears, draw a chord to that end.

This handles all of the left reflex vertexes

If we sweep the line right to left, we get rid of all right reflex points.

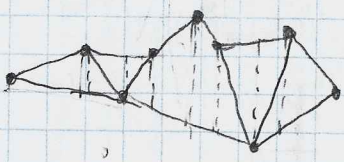
This however isn't sufficient to triangulate the polygon.

However, it is sufficient to make all the remaining polygons x-monotone & we can triangulate an x-monotone polygon in linear time.

Def:
A polygon is x-monotone iff every vertical line intersects w/ polygon at most twice.

Let's go over triangulating an x-monotone polygon in linear time.

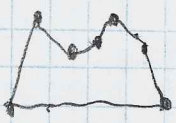
Problem:
Given some x-monotone polygon



Our first step is to draw chords b/w every time we alternate b/w top & bottom vertexes.
These are in a sense "easy".

Now, every polygon left is either a triangle or one side only has one edge.

This leaves us only w/ "comb" style polygons to handle.



Now, to triangulate these, we run Graham scan to find the lower hull (or upper if the top has 1 edge) where we go left to right.

The chords for the triangulation are the chords we pop off the stack.

We know this is correct b/c we need $n-3$ chords & the Graham scan pops off $n-2$ chords (b/c the lower hull is 1 edge) or $n-3$ not counting the bottom edge. Further, we know no chord intersects another.

More Sweeplines (& Event-based Algorithms)

Problem: Simple Polygon?
Determine if a polygon is simple (i.e. has no self-intersections).

Idea: Sweepline & (Planar)
Sweep a line across the polygon. Whenever a polygon edge gets added, see if the added vertex intersects w/ the edge of its neighbor above & below. Likewise, when a polygon gets removed, see if its previous

neighbors (who are now themselves neighbors) intersect.

If any edge intersects, bail out.

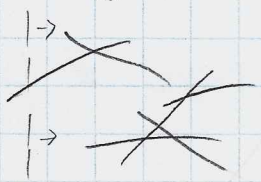
This runs in $O(n \log n)$ time b/c it takes $O(n \log n)$ to sort the data & you do $O(\log n)$ operations (O(1) check intersect & $O(\log n)$ insert/delete) on n edges.

Problem: Count Intersections

Given a bunch of sticks laying on the ground, how do you count the number of intersections?

Idea: Sweepline & Event Queue

Let's try to apply the sweepline idea again.



We run the sweepline across like last time but now we have to deal w/ crossings as well, so our events are

- insert
- delete
- cross

Then we just have our binary tree/sorted list from earlier. Also we store events in a priority queue.

Let's analyze the runtime. Let

n = number of vertexes

k = # of intersections $\leq \binom{n}{2} \in O(n^2)$

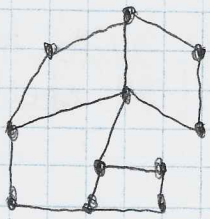
We have $2n+k$ events which each take $\log(n)$ time to handle if we're using a $O(\log(2n+k)) = O(\log(n))$ priority queue (i.e. heap). Thus the runtime is

$O((n+k)\log(n))$

Note: This is worse than the naive n^2 solution in the worst case of $k = \binom{n}{2}$.

Half Edges

Polygonal complexes are a set of polygons which share some vertexes & edges.

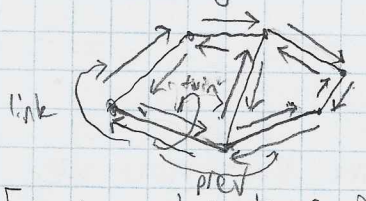


Note: We call a collection of half edges a half edge loosely

These appear in maps & Voronoi diagrams & other places.
How do we represent this?

The answer is half edges. Half edges are the building blocks of polygonal complexes & many other interesting geometric objects.
It's a neat datastructure!

Def:
A half edge is a collection of vertexes, edges, & (implicitly) polygons.



This is a collection of half edges

Every edge has 2 sides & each half edge has a
- next,
- previous, &
- twin

half edge.

The twin always points in the opposite direction & is on the opposite side.

In the 1D case, this is like a doubly linked list in form

Problem:
Iterate over all of the outgoing half-edges from a vertex, V, given a half edge h incident to vertex V

Solution:

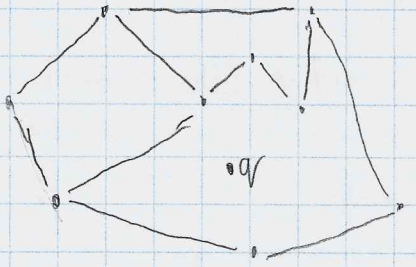
```
curr = h
while True:
    yield curr
    curr = curr.twin.link   # or curr = curr.prev.twin
    if curr is h:
        break
```

clockwise

counter-clockwise

Point Location in a Planar Subdivision

Point location in a planar subdivision is the process of locating a point in a bunch of half-edges.



Planar Straight Line Graph (PSLG)

Problem:
We want to convert PSLG into a query-able form about what region/face a point is in.

Input: PSLG & a query point q
Output: The face containing q (any half-edge on the face)

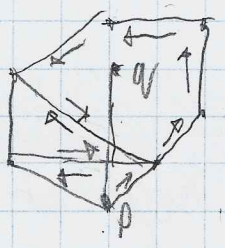
We use this by constructing a queryable datastructure.

Algorithm 1: Ray Casting
If we just cast a ray from q & find the closest edge intersecting the ray, we get what face q is in.

This is fairly naive but is still O(n), since finding the min takes linear time.

Algorithm 2: Walking
Let's start at a known point p & draw a line from p to q. Walk around your current face until you reach an edge that intersects the line. Then you switch to the twin of that half edge & walk around the face. Repeat this until you reach a face w/ no intersections. This face contains q.

Note: If you ever go to the outside of the polygon, you're done & it's outside the polygon.
Note: You should always start at a point on the outside edges.



This is O(n) in general, doing better in some cases.

Rem:
The Euler characteristic of the plane is 2. That is no matter what PLSG (informally shape) you draw
$n - m + f = 2$ where the PLSG has n vertices, m edges, & f faces.

Further, if all faces are triangles then $m = 3n - 6$.
If all inner faces are triangles, then $m \leq 3n - 6$.

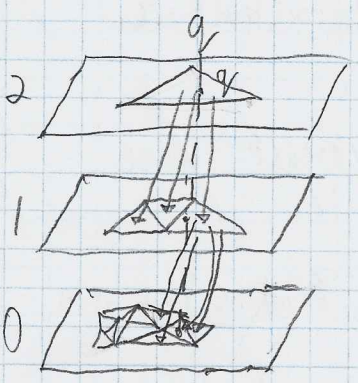
We know (from Don) we know this problem can be solved in $O(\log N)$ time. Tho that's difficult for this problem.

Let's try the simpler case where all faces are triangles.

Let's imagine we can search a smaller but somehow similar triangulation/PLSG that will help us know where q is.

To make this specific, let's split this problem into levels. We create $O(\log(n))$ levels where each triangle at level n has connections to all the faces it intersects w/ below.

At the top, we search an incredibly coarse grained PLSG. Then we go down a level & find a finer grained PLSGs & start w/ one of the child faces (& only search them).

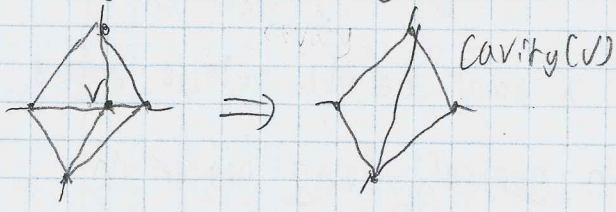


These arrows b/w faces forms a DAG.

Our goal for these layered triangulations is for them to be
i) shallow $O(\log n)$ OR i) Height
ii) bounded degree (so you don't expand too much) iii) Width

We'd also like to be able to easily construct it from the bottom level.

We can do this by cutting out individual vertices. This creates a cavity which you then triangulate.



n vertices $n-1$ vertices

If we always remove low-degree vertices this is bounded degree but it is not shallow w/ height n.

We're going to have to remove a lot of vertices at a time (half of them) to reach $O(\log n)$ height. We can remove multiple vertices where no two are adjacent (i.e. independent set).

We want to remove an independent set of "low degree" vertices (i.e. below some constant).

We know such vertices exist b/c the graph is planar so the number of edges m is $m \leq 3n-6$ where n is the number of vertices. Formally

$$m \leq n-6$$

$$\sum_{v \in V} \deg(v) = 2m \Rightarrow \frac{1}{n}\sum_{v \in V} \deg(v) = \frac{2m}{n} \leq \frac{6n-12}{n} < 6$$

Therefore at least some vertices have degree <6. Let's show $\frac{1}{2}$ have some low degree.

In general, you can't have 50% things 2x above average

Lemma:
In a triangulation, at least $\frac{1}{2}$ vertices have degree <12.

pf:
Suppose for contradiction $\exists \frac{1}{2}$ vertices w/ deg ≥ 12

$$m = \frac{1}{2}\sum_{v \in V} \deg(v) \geq \frac{1}{2}\left(\frac{1}{2} \cdot 12\right) = 3n$$

This is a contradiction, so at least $\frac{1}{2}$ have degree <12.

We know at least $\frac{1}{24}$ are degree <12 & independent. This is b/c in the worst case we lose 11 of our vertices of the $\frac{1}{2}$ w/ deg <12.

Thus, our greedy independent set of vertexes has at least $\frac{1}{24}$ vertexes.

Let's analyze the runtime.

Analyze Algorithm 3:
Let $n_0 = \#$ points in level i

$n_0 = n$

$n_{i+1} = \left(\frac{23}{24}\right) n_i$ so $n_h = \left(\frac{23}{24}\right)^h n$

Since $n_h = 1$, we have

$$n = \left(\frac{24}{23}\right)^h$$

so $h = \log_{\frac{24}{23}} n_0$

This is our shallow & low degree levels.

Note: This is actually worse in practice ✓ normally b/c it requires building the datastructure & $\log_{\frac{24}{23}} n$ isn't very good. Linear time w/ no updating/building of the datastructure makes the walking very cheap.

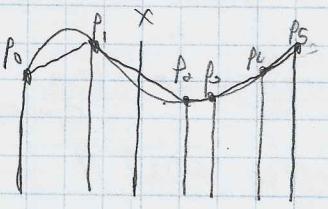
Let's go over the preprocessing costs of

- $O(n \log n)$ to eat triangles
- $O(n_i)$ to find independent set $= \sum_{i=0}^{?} n_i = \sum_{i=0}^{?} \left(\frac{23}{24}\right)^i n = O(n)$
- $O(n)$ for removal & retriangulation

$\Rightarrow$ Total is $O(n \log n)$ time to build data structure. (But $O(\log n)$ to search)

Interpolation of Functions & Delaunay Triangulation

The simplest case is linear interpolation of a function $f: \mathbb{R} \to \mathbb{R}$ (or maybe more appropriately a piecewise affine combination). You have a function w/ some points you know the value of



The function is $f(x)$. Our piecewise approximation is $\hat{f}(x)$.

Given a point x inside the range of known values, we find the two points that most tightly enclose the point x. Here it's p_1 & p_2.

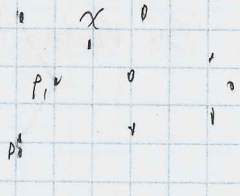
We then write x as an affine combination of p_1 & p_2. That is

$x = \alpha p_1 + \beta p_2 = \alpha p_1 + (1-\alpha) p_2$

so we know

$\hat{f}(x) = \alpha \hat{f}(p_1) + (1-\alpha) \hat{f}(p_2)$.

Let's consider extending this to a 2D function $f: \mathbb{R}^2 \to \mathbb{R}$ where we know a cloud of points.



The points are labelled arbitrarily.

Now, to f at x that is find $\hat{f}(x)$, we find 3 points a, b, c which enclose x. Then we write x as an affine combination of a, b, c to find $\hat{f}(x)$

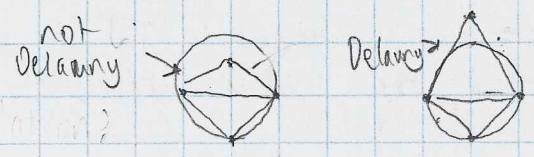
$x = \alpha a + \beta b + \gamma c$ so $\hat{f}(x) = \alpha \hat{f}(a) + \beta \hat{f}(b) + \gamma \hat{f}(c)$ where $a, b, c \geq 0$ & $\alpha + \beta + \gamma = 1$.

How do we pick good points a, b, c? We want close points that are evenly spaced. That is we want triangles near equilateral.

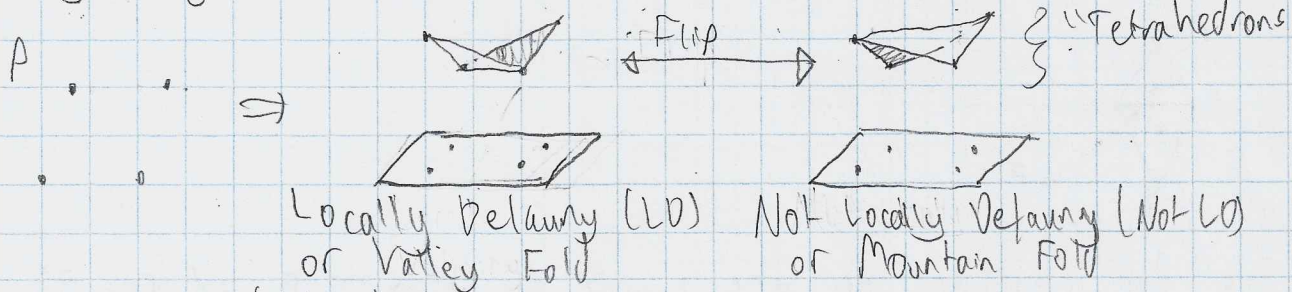
How do we find this? The trick is to find a Delaunay triangulation of the point cloud A. That a triangulation of P of only Delaunay triangles

Def:
Consider some point cloud $P \subset \mathbb{R}^2$. $Del(P)$ is a triangulation of P such that for all triangles $T \in P$ the interior of the circumcircle of T does not intersect w/ P.

That is, given a triangle $\triangle abc \in P$ & another point $d \in P$
$d \notin \text{incircle}(\triangle abc, d)$



We find this Delaunay triangulation by raising all of the points onto a paraboloid centered at an arbitrary origin. Then we find the lower convex hull.



A triangulation is (globally) Delaunay if all points are locally-Delaunay & we can swap a "tetrahedron" from LD to not LD or vice-versa.

This gives us an idea on how to make an arbitrary triangulation. Just keep flipping the tetrahedrons to be LD.

Thm:
The Delaunay triangulation for a point cloud $P \subset \mathbb{R}^2$ exists iff it is in <u>general position</u>, that is no 3 points are colinear & no 4 are cocircular

Example:
A square does not have a Delaunay triangulation b/c the 4 points are cocircular.

Problem:
Find the Delaunay triangulation of a point cloud $A \subset \mathbb{R}^2$ in general position.

Algorithm ?:
Let's follow our nose on the flipping idea.

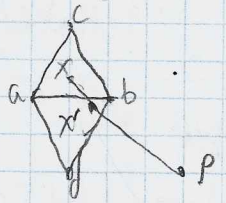
while P has not LD tetrahedrons:
    Flip the tetrahedron

This is a problem tho. How do we know we always make forward progress? Also how do we know we can always flip not 4D tetrahedrons?

We will argue that flipping always makes forward progress. The idea is that are add simplexes (tetrahedrons in 3D) whenever we flip. Essentially the bottom of the convex hull keeps having dents hammered out.

Thm:
If our triangulation T of point cloud P has edges are locally Delauny, then the entire triangulation is Delauny. That is
$T = Del(P)$.

pf:
Consider some point $p \in P$ & any $x \in conv(P)$ ← convex closure of P. Let Δabc be the triangle containing x.



We do induction on the number of crossings k w/ ray $\vec{xp}$.

In our base case of $k=0$, then $p \in \{a, b, c\}$, so $p \notin$ circle(a, b, d).

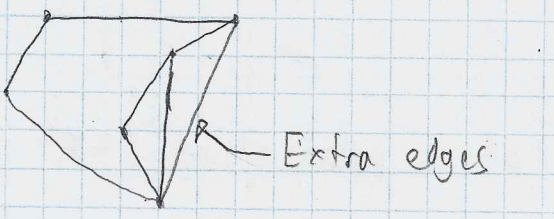
Now th. inductive case assume $k>0$ ∨ $p \notin$ circle(a, b, d) where Δabd has $k-1$ crosses.
By induction $p \notin$ circle(a, b, d).

Thus the triangulation T is Delauny.

Now, our algorithm depends on a triangulation. How do we find a triangulation?

Problem:
Given a point cloud P. Find a triangulation T.

Algorithm 1: Complex
First find a simple polygon inside of the point set. This takes $O(n \log n)$ time.
Then we triangulate this simple polygon, which takes $O(n \log n)$ time.
Then were just left w/ some points on the outside w/ no edges.



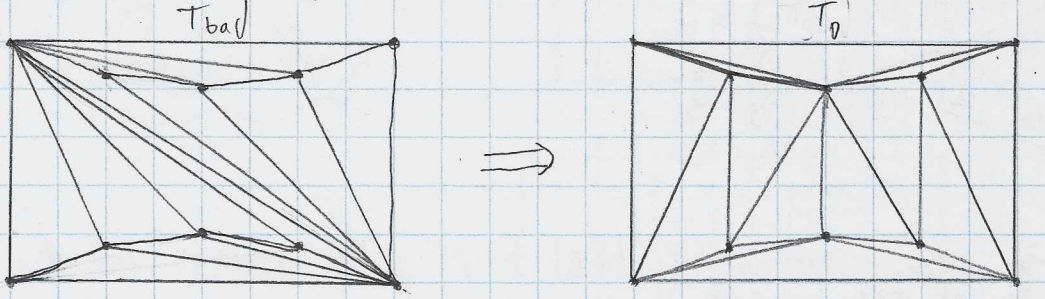
To get this run Graham scan along the outside & keep every edge you make.

Problem:
Given a triangulation T of a point set P. Transform the triangulation T into a Delaunay triangulation T_D.

Before we try to make an algorithm, let's do an upper bound.

We know the lower bound is at least $O(n^2)$ flips. This corresponds to you only "permanently flipping" one edge. That is you always make a progress of at least 1 every iteration over the n edges.

Here is an example of a worst case triangulation.



"Flip to Delaunay Algorithm"

Here only one edge can be flipped, the long edge in the middle. So we have to do quadratic $O(n^2)$ flips.

Note: Just b/c some triangulations need $O(n^2)$ flips, we can still find the Delaunay triangulation in $O(n \log n)$ time. We just need good triangulations. (Or incremental/clever design.)

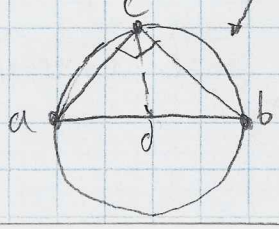
Note: Delaunay triangulations & their algorithms as described easily generalize to higher dimensions.

Def:
The max-min property of triangulations of a point set P states that the minimum angle of all the triangles is maximized.

Thm:
The Delaunay triangulation satisfies the max-min property.

Note: There may be other triangulations that satisfy the same property

Thm: Thale's Theorem
If $\overline{ab}$ is the diameter of a circle & c is on the circle, then $\angle acb$ is 90°



You can prove this by drawing $\overline{cd}$ & using isosceles triangles

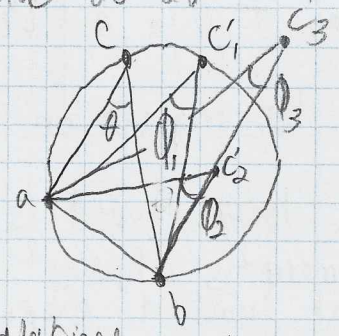
Lem: Generalized Thale's Theorem

Given a chord $\overline{ab}$ of a circle & point along the circle c.
Let c' be a point on the same side as $\overline{ab}$

$\angle acb = \angle ac'b \Rightarrow c'$ on circle

$\angle acb < \angle ac'b \Rightarrow c'$ outside circle

$\angle acb > \angle ac'b \Rightarrow c'$ inside circle

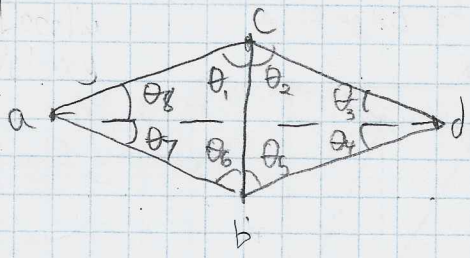


$\phi_1 = \theta$

$\phi_2 > \theta$

$\phi_3 < \theta$

This generalization of Thale's theorem gives us nice facts about Delaunay triangulations.



$\theta_1 > \theta_4$
$\theta_2 > \theta_7$
$\theta_5 > \theta_8$
$\theta_6 > \theta_3$

It's unknown whether you can flip b/w two 3D triangulations

With some work, you can show that flips will always reduce the minimum angle of the triangle.

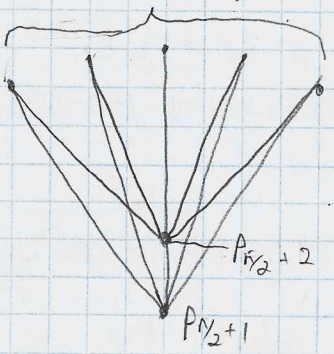
This thus proves the max-min property of Delaunay triangulations. □

Proving Flip-to-Delaunay is $O(n \log n)$

Our idea is to incrementally build a Delaunay triangulation by having some points in a Delaunay triangulation, adding 1, & then doing minimal modifications to the triangulation to make it Delaunay.

If at each step we do at most $\log(n)$ flips at every insertion, we're $O(n \log n)$.

Sadly, this isn't true for any way to insert points. Here's a counter-example



This requires

$n/2$ flips for $P_{n/2+1}$

$n/2$ flips for $P_{n/2+2}$

...

Here, we do a linear number of flips at each point, so it's $O(n^2)$.

But this case feels really unlikely. What if we used some randomization. Could we then have expected time $O(n \log n)$? If so, how do we prove the runtime?

Spoiler: We can.

This method of randomly building this incremental Delaunay triangulation is called RIC.

To prove that RIC's expected runtime is $O(n \log n)$, we do backward analysis.

In backward analysis, instead of starting w/ nothing & computing all the possibilities, you start from the solution & compute what your previous steps might have been.

Let $p_1, \ldots, p_n$ be some permutation of a pointset P.
Let Q_i be the point set at step i.

We want to find $\deg(p_i)$ in Del_{Q_i} to find the number of flips we might have to do for p_i. Call $\deg(p_i) = \delta_i$.

Let N be the total number of flips as a random variable

$$E[N] \leq E\left[\sum_{i=1}^{n} \delta_i\right] = \sum_{i=1}^{n} E[\delta_i] \leq \sum_{i=1}^{n} 6 = 6n$$

We know $E[\delta_i] = 6$ by Euler's formula in a plane.
Let v, e, f be the number of vertexes, edges, & faces.
We know the Euler characteristic of a planar subdivision is 2 so

$$v - e + f = 2 \quad \& \quad e = 3/2 f$$

$$\Rightarrow v - e + 2/3 f = 2$$
$$\Rightarrow 3v - 3e + 2f = 6$$
$$\Rightarrow e = 3v - 6$$

The total degree is $2e$ the # of edges b/c each edge is counted twice

$$E[\delta_i] = \sum_{i=1}^{n} \delta_i \cdot \tfrac{1}{n} \quad \text{(b/c random)} = \frac{\sum_{i=1}^{n} \delta_i}{n} = \frac{2e}{n} = \frac{6v - 12}{n} \quad (v = n \text{ b/c we're counting points (vertexes)}) = \frac{6n - 12}{n} < 6$$

Delaunay Triangulations w/ Edges

Thm:
Given point set P w/ $a, b \in P$. We know edge $\overline{ab} \in Del(P)$ iff $\exists$ a circle w/ a, b on its boundary that contains no points of P.

Def: Gabriel Graph
Given a point set P, the Gabriel graph (P, E) is a graph where $(a, b) \in E$ iff circle w/ diameter $\overline{ab}$ contains no points of P.

Thm:
Given point set P, Gabriel$(P) \subseteq$ Del(P).

Def:
Given some point set $P \subset \mathbb{R}^2$, the Euclidian minimum spanning tree (MST) $T = (P, E)$ minimizes $\sum_{ab \in E} \|a - b\|$.

Every MST edge is Gabriel

Thm:
The MST of a point set is a subset of the Gabriel graph.

Pf:
Suppose for contradiction that MST $\not\subseteq$ Gabriel graph. Then there exists an edge $\overline{ab}$ where the diameter contains some point c.

Since the MST is a tree, if you remove $\overline{ab}$ then c is still connected to either b or a.

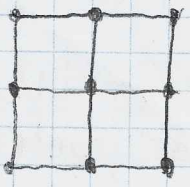
WLOG, suppose c is connected to b. Then if we swap $\overline{ab}$ w/ $\overline{bc}$ we have a new smaller spanning tree, meaning our original wasn't the MST.

Thus MST $\subseteq$ Gabriel graph.

Coro:
The Gabriel graph is always connected.

Thm:
The converse of the theorem is not true.
So Gabriel graph $\not\subseteq$ MST.

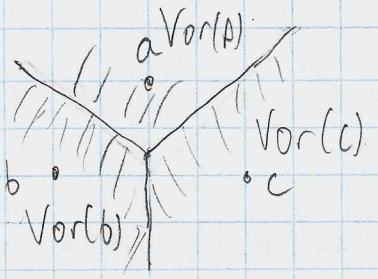
Pf:
Consider a grid of points.



It's Gabriel graph is clearly not a tree.

Voronoi Diagrams

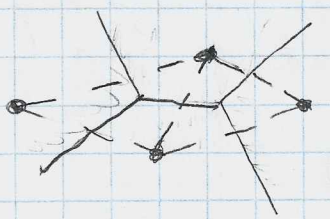
Def:
Given a set of points $P \subset \mathbb{R}^2$, find a subdivision of $\mathbb{R}^2$ in voronoi cells where for each $p \in P$
$Vor(p) = \{ x \in \mathbb{R}^2 \mid d(x,p) \leq d(x,q) \ \forall q \in P \}$.
& equivalently $\|x-p\| \leq \|x-q\|$



Thm:
The dual of the Voronoi diagram is the Delauny triangulation.

Alternatively, the Voronoi diagram & Delauny triangulation are dual.

Example:



-- Delauny triangulation
— Voronoi diagram

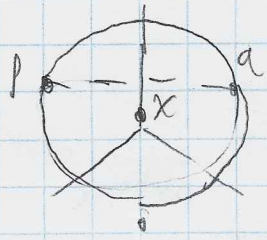
We propose that we can lossly switch b/w Delauny & Voronoi
point ⇔ cell
edge ⇔ edge
triangle ⇔ corners (circumcenter)

Delauny	Voronoi

Let's show we can go from edges to edges.

Suppose we have some $x \in Vor(p) \cap Vor(q)$ for some $p, q \in P$ w/ $p \neq q$.

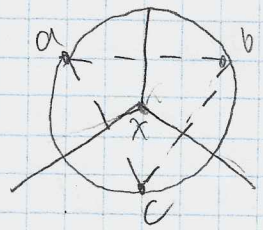
We know that $d(x,p) = d(x,q)$ & circle$(x, d(x,p))$ is empty.
& x is the center



And so we conclude that $\overline{pq} \in Del(P)$.

Let's show triangles $\leftrightarrow$ corners.

Consider some $x \in Vor(a) \cap Vor(b) \cap Vor(c)$



By similar logic we know $r = d(x,a) = d(x,b) = d(x,c)$ & so x is the center of the circle formed by $\triangle abc$

$\Leftarrow circle(\triangle abc) = circle(x, r) = C$.

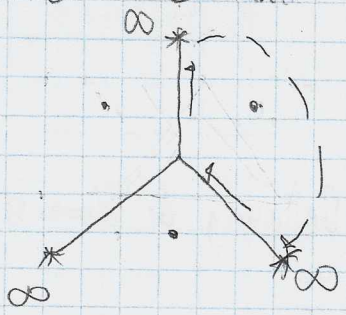
Suppose there exists a $d \in C$. Then $d(x,d) < d(x,d)$, which is impossible b/c we assume $x \in Vor(a)$.

Thus there are no points in C & so x is the circumcenter of the Delaunay triangle $\triangle abc$

Voronoi Diagram & Half Edges

Naively, it seems like we can't represent a Voronoi diagram w/ half-edges.

However, if we add a single point ∞ to $\mathbb{R}^2$ then you can make all of those edges going to ∞ have a vertex at ∞. Then we connect the half-edges up w/ their other half-edge going to ∞.



(All ∞'s are the same)

You can visualize this as wrapping space around onto a sphere. This is called the 1-point compactification of the plane.

Thm:
Voronoi cells are convex. Further, they are polyhedrons.

Polyhedrons & Linear Equations

A linear equation is of the form

$$ax + by = \begin{bmatrix} a \\ b \end{bmatrix}^T \begin{bmatrix} x \\ y \end{bmatrix} = c$$

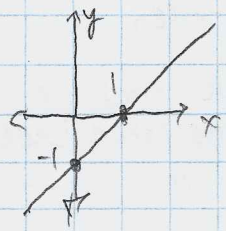
or more generally

$$v^T x = c$$

v: fixed, x: variable, both $\mathbb{R}^2$

The set of solutions $\{x \mid v^T x = c\}$ is a hyperplane. ($D-1$ dimensional subspace)

Consider the solutions to $x-y=1$



Note: All hyperplanes can be written in this form ↓

Let $H=\{x \mid v^Tx=c\}$ be a hyperplane.
The vector v is the normal vector of H, that is it is orthogonal to every basis vector if H. In other words

Thm:
Let v be a normal vector to the hyperplane $H=\{x \mid v^Tx=c\}$.
Then $v(a-b)=0$ for all $a,b \in H$.

Note: $ax+by=c$ is the best way to write an equation :). (I'm not sure ± believe this.)

Pf:
$$v^T(a-b)=\underbrace{v^Ta-v^Tb}_{a,b\in H}=c-c=0$$

Def:
A half-space is the set of solutions to a linear inequality

$H=\{x \mid v^Tx\leq c\}$ ← Half-space

$\partial H=\{x \mid v^Tx=c\}$ ← Hyper-plane

Thm:
All half-spaces H are convex.

This can be proven by showing given any H is closed under convex combination.

We have no bound on n. You can have 0 or 1 half-spaces & we count it as a polyhedron.

Def:
A polyhedron is the intersection of a finite set of half-spaces.

$$P=\bigcap_{i\in[n]} H_i \quad \text{where} \quad H_i=\{x \mid v_i^Tx\leq c_i\}$$

Or $P=\{x \mid v_i^Tx\leq c_i \ \forall i\in[n]\}=\{x \mid Vx\leq c\}$
where $V=\begin{bmatrix}\text{—}\ v_i^T\ \text{—}\\ \vdots\end{bmatrix}$ & $c=\begin{bmatrix}c_i\\ \vdots\end{bmatrix}$

Concern, what if we want some to be $\geq$? multiply by -1.

Also $\leq$ is not well defined on $\mathbb{R}^n$ so we define it element-wise.

Notation: Often P is shortened as $Vx \leq c$

I wish we had an explicit broadcasting/element-wise notation

Thm: All polyhedrons are convex. This falls immediately from the following theorem.

Thm: The intersection of convex sets is convex.

Def: A set $X \subseteq \mathbb{R}^d$ is <u>bounded</u> iff $\exists r > 0$ s.t. $X \subseteq B_r(0)$. That is the set can be contained w/in a finite ball (around the origin).

Def: A <u>polytope</u> is a bounded polyhedron.

Thm: The only unbounded polyhedrons in a Voronoi diagram are the cells about the points on the convex hull.

Thm: Given a D-dimensional space X, you need at least $D+1$ half-spaces in a polyhedron to form a polytope.

Def: A simplex in D-dimensional space X is a polytope w/ $D+1$ dimensions.

It is the <u>simplest</u> polytope in that space.

Thm: Given finite point set $P \subseteq \mathbb{R}^d$. $\text{conv}(P)$ is a polytope.

Thm: The main theorem of polytopes
S is a polytope iff $\exists$ finite point set $P \subseteq \mathbb{R}^d$ s.t. $S = \text{conv}(P)$.

Voronoi & Delaunay

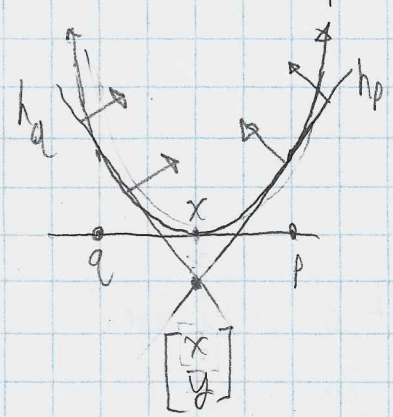
Let $P \subseteq \mathbb{R}^d$ be a point set. Let $P^+ = \left\{ \begin{bmatrix} p \\ \|p\|^2 \end{bmatrix} \mid p \in P \right\} \subseteq \mathbb{R}^{d+1}$ be a parabolic lifting of P.

Let $H = \{ h_p \mid p \in P \} \subseteq \mathbb{R}^{d+1}$ where $h_p = \left\{ \begin{bmatrix} x \\ z \end{bmatrix} \mid z \geq 2p^T x - \|p\|^2 \right\} \subseteq \mathbb{R}^{d+1}$ (hyperplane)

Previously, we showed Del_P = lower projection of $\text{conv}(P^+)$.

Dually, Vor_P = projection of $\bigwedge_{p \in P} h_p$.

Consider an example



Note: We say $\dim(\emptyset) = -1$

Suppose $\begin{bmatrix} x \\ y \end{bmatrix} \in \partial h_p \cap \partial h_q$, that is

$$y = 2px - p^2 \quad \& \quad y = 2qx - q^2$$

So

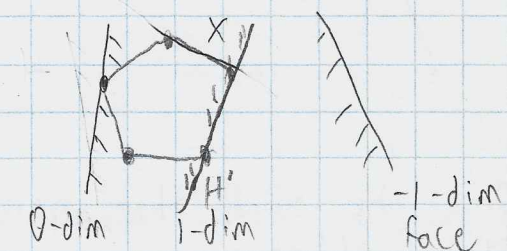
$$\begin{aligned} 2px - p^2 &= 2qx - q^2 \\ 2px - 2qx &= p^2 - q^2 \\ 2x(p-q) &= (p+q)(p-q) \\ x &= \frac{p+q}{2} \end{aligned}$$

So x is the mid-point, which is where the Voronoi-diagram's edge is. Awesome!

Faces of a Polyhedra

Def:

A supporting hyperplane H of polyhedron X st H bounds a halfspace containing X.



Def: Face Face

A face of $_{\text{polyhedron}}X$ is $X \cap H$ for some supporting hyperplane H.

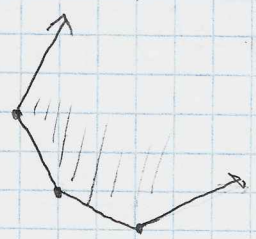
We can express Eulers formula

$v - e + f = 2$ for polygons

generalizes to

$n_0 - n_1 + n_2 = 2$ where $n_i = \#$ i-faces

Consider the following 2-polyhedron



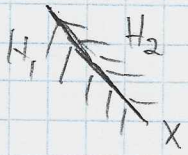
$$\begin{array}{ccccc} n_0 & - & n_1 & + & n_2 \\ \| & & \| & & \| \\ 3 & - & 4 & + & 1 = 0 \end{array}$$

Rem:
Prove every face face of a polyhedron is itself a polyhedron.

This is b/c a polyhedron is an intersection of half-spaces & a face is an intersection of a polyhedron & a half-space, that is an intersection of half-spaces itself.

Coro:
A hyperplane X is a polyhedron. We define H_1 & H_2 as the two opposite side planes of X.

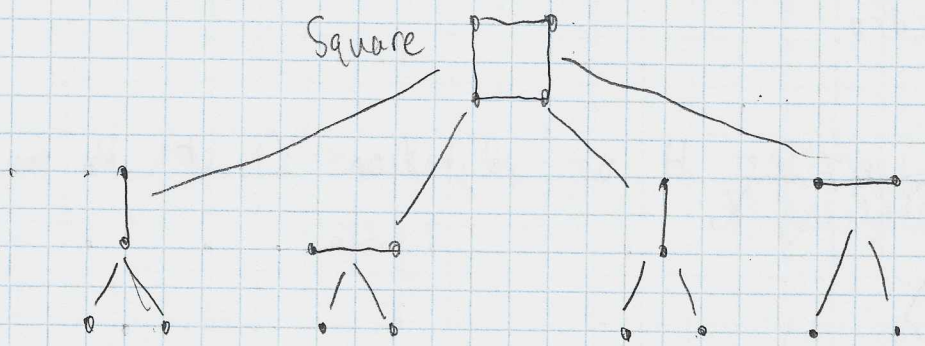
$X = H_1 \cap H_2$



We can define a partial order of polyhedrons. We can do this by taking some polyhedron. It is made up of faces, edges, etc, which are themselves polyhedrons.

We say these sub-parts as smaller than their parent.

This gives us a tree called a <u>lattice</u>.



Linear Programming

Linear programming is the process of finding the point that maximizes some linear equation, called the objective function.

For example, we want to find x such that
$\max c^T x$ where $Ax \leq b$.

Let's write an <u>incorrect</u> solution to the travelling salesman problem (TSP) using linear programming.

Problem: TSP

Given a graph $G=(V,E,w)$ where $w: E \to \mathbb{R}^+$ is the weights (w/ V & E vertexes & edges), find a cycle of length $n=|V|$ in G that minimizes

$$\sum_{e \in Cycle} w(e).$$

Algorithm:

We define variable x_e for each $e \in E$ where

$0 \le x_e \le 1$ where $\sum x_e = n$.

This corresponds to the restriction of hitting all edges.

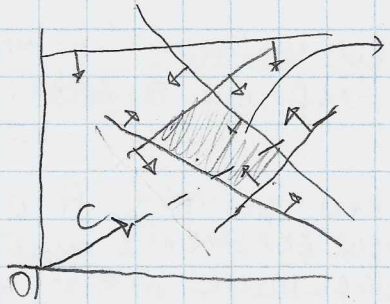
Further, we enter & exit every vertex, that is the sum of its neighboring edges is 2. Thus

$$\sum_{u \in V} x_{ru} = 2 \quad \forall x \in V.$$

The issue w/ this formulation is we say $0 \le x_e \le 1$. that is we allow the notion of taking "half" an edge. To solve this we must say $x_e = 0$ or $x_e = 1$. but that is an integer linear program, which is NP complete.

We wanted a polynomial time algorithm.

Let's consider the following 2D linear program



We define a feasible region $\{x \mid Ax \le b\}$.

We define the maximum as a vector c & we just pick the farthest point in the feasible region:

$$\max c^T x = \max \frac{c^T x}{\|c\|}.$$

How do we solve such a system? We'll cover the simple method. The idea is you start on an edge of the feasible region (called a tight constraint) & then walk around the edges until you reach the maximal one.

Let's walk thru solving a 2D linear program. (See it as a pile of sticks)

Algorithm 1:

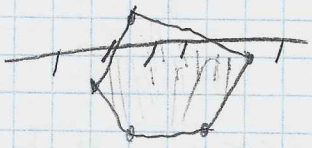
Suppose we have m constraints. We first try to find the polygon of the feasible region.

In the worst case, we have m edges & vertexes.

The naive solution is to go thru all vertices & see if they satisfy all constraints. This is $O(m^3)$.

Algorithm 2: Random Incremental Linear Programming in 2D

Suppose we have some feasible polygon which we're adding a new constraint to.



How do we know what new vertexes to add? You go down a dimension & consider just the 1D case where you're solving the problem along the new dimension.

Remark:

In general, linear inequalities are much easier to solve than equalities. For example, in 2D you need 2 (independent) lines, if you have more they either provide no new information or make it inconsistent.

Algorithm 2 cont: RI LP in 2D

Revisiting the algorithm, when we add a new constraint, if the current optimum satisfies the new constraint, then we do nothing, so $O(1)$ time.

If the current optimum doesn't satisfy the new constraint, then we search for the new optimum along the new-constraint's hyperplane, by dropping down a dimension which takes $O(k)$, $\hat{=} 10$ time where k = # of constraints currently

We know the new constraint is tight b/c if it wasn't then the old optimum would have worked.

Here is the pseudo-code.

randomly order constraints
find solutions $x_1, \dots, x_n$
at step k:
  if x_k fails new constraint (i.e. $A_k^T a_k > b_k$):
    reduce problem by 1 dim

Let's analyze the runtime of the algorithm.

Analysis of Algorithm 2: In $\mathbb{R}^2$

At each step, there are only 2 constraints, so the probability that the previous step was tight is $2/k$ while it not being tight is $\frac{k-2}{k}$.

$$\sum_{k=1}^{m} \underbrace{\frac{2}{k} O(k)}_{\text{tight case}} + \underbrace{\frac{k-2}{k} O(1)}_{\text{not-tight}}$$

$$= \sum_{k=1}^{m} O(2) + O\left(\frac{k-2}{k}\right)$$

$$= \sum_{k=1}^{m} O(1)$$

$$= O(m).$$

So the expected runtime is linear in number of constraints in 2D.

A similar analysis applies to higher dimensions & indeed the runtime is always linear for constant dimensions.

However, if you let dimension d vary, you will see that the constant $O(1)$ in the sum becomes $O(d!)$, giving us overall runtime of

$$O(m(d!))$$

Exercise:

Given some hyperplanes $H_1, \dots, H_n$ w/ normal vectors $v_1, \dots, v_n$, see if the hyperplanes define a polytope (i.e. are bounded).

This happens iff $\exists \alpha_1, \dots, \alpha_n \in \mathbb{R}^n$ such that

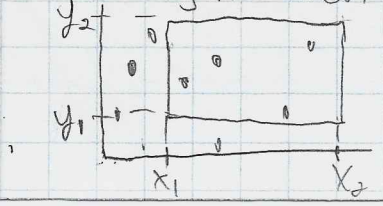
$$\alpha_1 v_1 + \dots + \alpha_n v_n = 0.$$

Intuitively, iff you can draw a cycle w/ the normal vectors, then it's a polytope.

Searching Datastructures

kd-Tree

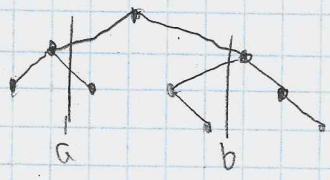
Given n points in $\mathbb{R}^d$, we want to efficiently find all points in a (generalized/higher dimensional) rectangle. This should work for any query.



This is called orthogonal range searching b/c it's a rectangle that is lined up w/ the axis.

We're going to build a kd-Tree, which is a generalization of binary trees.

If we do the 1D version of binary trees, where we want to find all elements b/w a & b

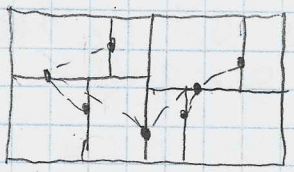


There are n nodes & k w/in the range.

To do this, you search for a & then you search for b ($O(\log n)$). While you traverse to these nodes, you add every term w/in the range ($O(k)$). This gives us a runtime of $O(k + \log n)$.

This is the best we can do.

Let's extend this to 2D. To start, given a bunch of points, draw a vertical line thru the median. Then draw horizontal lines in the median of the 2 sides. Then vertical lines in the remaining 4 rectangles.



The idea here is to start at the root. If it's to the left of the search area, prune the left. If it's to the right, prune the right. Add the point if it's in the range & recurse to all unpruned children, flipping axis/coordinate considered. In higher dimensional cases, you cycle thru the axises.

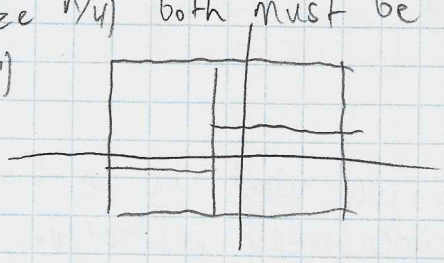
Let's analyze the runtime. At each level, we have to find the median. We can do this in $O(n)$ time if we're clever. Here is the recursive analysis

$$T(n) = O(n) + 2T(n/2) = O(n \log(n)).$$

Thus the kd-tree takes $O(n \log(n))$ construction.

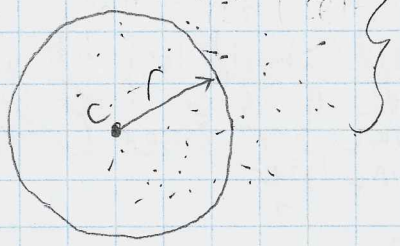
Let's analyze runtime for queries. We need to find the maximum number of crossings we see. Each iteration you have at most 2 crossings w/ rectangles. Those two rectangles (of size $n/4$) both must be recursed into so

$$C(n) = 2 + 2C(n/4) = O(\sqrt{n})$$



Circular Range Searches

How do we find points in balls rather than rectangles?
It's very similar to the rectangular problem. It just requires you to be able to find intersection b/w a ball & a box.



Cloud has kd-tree

```
def ballsearch(self, c, r):
    if dist(self.point, c) ≤ r:
        yield self.point
    if range ∧ self.left:
        yield from ballsearch(self.left, c, r)
    if range ∧ self.right:
        yield from ballsearch(self.right, c, r)
```

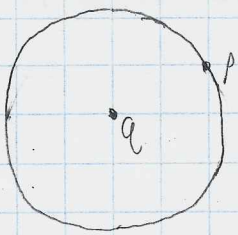
Could use method call syntax

It's left as an intersection to find efficient intersection algorithms.

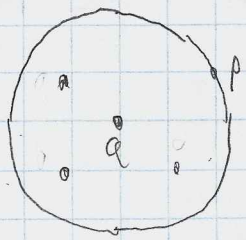
Nearest-Neighbor (NN) Search

Given a finite point cloud $P \subset \mathbb{R}^d$, build a datastructure that efficiently minimizes the distance b/w arbitrary query points $q \in \mathbb{R}^d$ & the points in the cloud $p \in P$.

This is related to searches in balls b/c if we have a point $q \in \mathbb{R}^d$ & point $p \in P$, we know p is q's nearest neighbor & we know if p isn't then the ones in the ball are the only possibilities.



p is q's NN



p is not q's NN

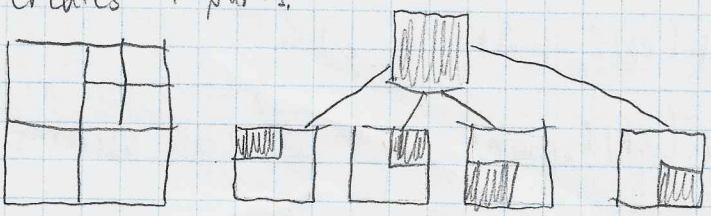
Here's some pseudocode for this.

```
def nnsearch(self, q, nbr=None):
    if nbr is None:
        nbr = self.point
    if dist(q, self.point) < dist(q, nbr):
        nbr = self.point

    if self.left and self.left.range ≠ ∅:
        nbr = self.left.nnSearch(q, nbr)
    if self.right ...:
        nbr = ...

    return nbr
```

{ Could/should look at whichever side q is on first b/c that will make nbr closer faster, saving time later.
Maybe...?

Quad Tree (QT)

This is only logically named in 2D, but essentially at each layer you split in half at each dimension. This creates 2^d children splits. So in 2D it creates 4 parts.



This is more purely geometric than kd-trees b/c you split areas.

Here's the construction:

while a square has >1 point: split it 4 ways

{ Note: This assumes no duplicate points or else this doesn't terminate

Since this deals w/ areas, you can't bound the height in terms of n. (Consider 2 arbitrarily close points.)

You can bound it by the spread.

Def:
The spread σ of a point cloud $P \subset \mathbb{R}^d$ the ratio b/w the diameter of P & the mesh of P, that is the max distance b/w 2 points & the min distance.

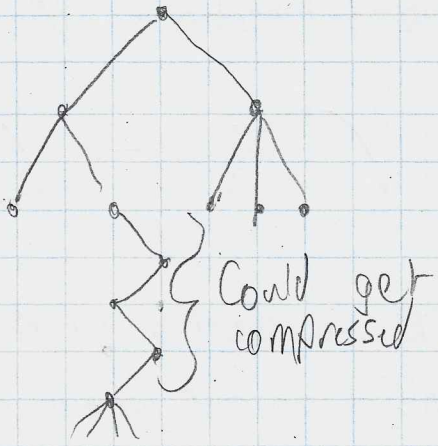
$$\text{spread} = \sigma = \frac{\max_{p,q \in P} d(p,q)}{\min_{p,q \in P, p \neq q} d(p,q)}$$

The height of the quadtree is bounded by spread

$$\text{height} = O(\log(\text{spread})) \iff h = O(\log \sigma).$$

We can compress a quadtree to get its size - in terms of n.

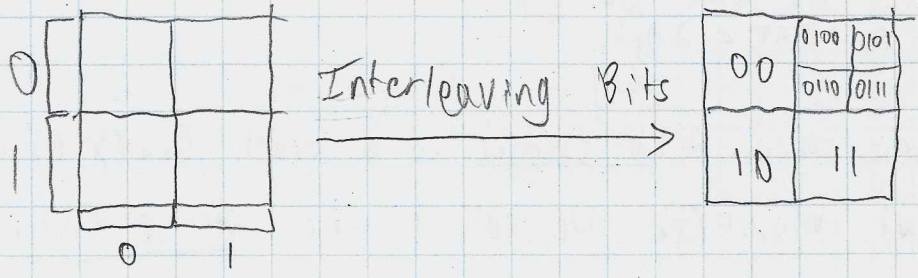
Compressing a quadtree is similar to compressing other kinds of trees, like tries. What you do is replace several non-branching edges w/ a single edge



Quadtrees are actually intimately related to tries & are often represented as such.

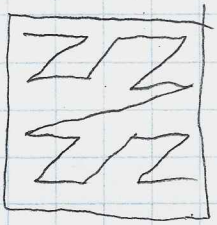
This gets the size of the QT down
Size: $O(n)$
Height: $\min(O(n), O(\log a))$

The trick to representing quadtrees as tries is to create tries using the bitstrings of their integer coordinates, where you first look at the MSB



This uses a space filling curve, which is a function $\mathbb{Z} \to \mathbb{Z}$. This function is our interleaving of the bits.

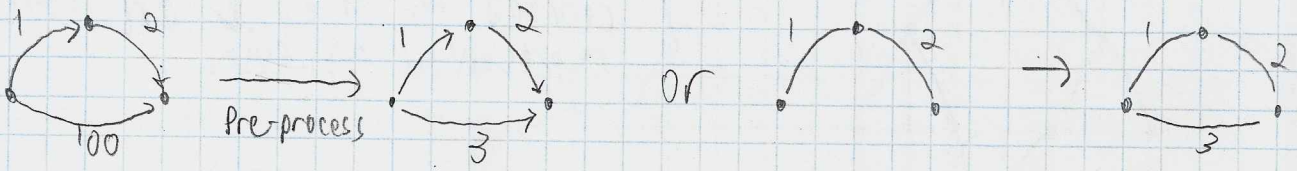
This space-filling curve gives a total order to the space & is called such b/c of this



Beyond $\mathbb{R}^0$

Def: A metric graph G is a graph where the distance b/w two nodes is a properly defined metric.

Often, when we are given a graph, we "preprocess" it by calculating the distance b/w all nodes including "shortcuts" b/w other nodes. For example



Let's do the travelling salesman problem (TSP) on metric graphs.

In particular, let's do a 2-approximation, that is we get w/in a factor of 2 of the optimal solution.

Algorithm: 2-approx TSP

Consider a metric graph G w/ minimum spanning tree MST. Let Opt be the optimal TSP tour.

By definition we know the $\text{len}(MST) < \text{len}(Opt)$ b/c the MST is not a tour.

Do a pre-order traversal of the MST to form the approx tour. We hit every edge twice, so we know $\text{len}(approx) \leq 2\,\text{len}(MST)$.

Now combining these two facts we get

$$MST < Opt \leq approx \leq 2\,MST < 2\,opt.$$

Def: Let (X, d) be a metric space. A t-spanner is a graph $G = (X, E)$ is a graph where

$d_G(u, v) \leq t\, d(u, v)$ $\forall u, v \in G$. We call t the stretch factor.

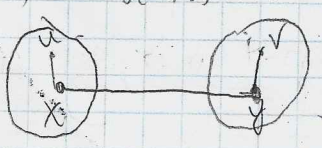
We want to use the spanner to approximate the metric space w/ a metric graph.

Our ideal spanner is low-degree & has a small stretch factor $t \approx 1$.

One generally good spanner for $\mathbb{R}^0$ is the Delaunay triangulation.

Consider metric space (X, d) & $t \in \mathbb{R}_0$. Let $\varepsilon = \frac{t-1}{2(t+1)}$.

Let $G = (X, E)$ where $\forall u, v \in X$ $\exists (x, y) \in E$ such that $d(y, v) \leq \varepsilon\, d(u, v)$ & $d(x, u) \leq \varepsilon\, d(u, v)$



} These balls both contain t-spanners

We claim G is a t-spanner. Consider some $u, v \in G$. We know

$$\begin{aligned} d_G(u,v) &\le d_G(u,x) + d_G(x,y) + d_G(y,v) \\ &\le t\,d(u,x) + d(x,y) + t\,d(y,v) \\ &\le t\,d(u,x) + d(x,u) + d(u,v) + d(y,v) + t\,d(y,v) \\ &= (t+1)\,d(u,x) + d(u,v) + (t+1)\,d(y,v) \\ &= 2(t+1)\,\varepsilon\,d(u,v) + d(u,v) \\ &= (2(t+1)\varepsilon + 1)\,d(u,v) \\ &= \left(2(t+1)\,\frac{(t-1)}{2(t+1)} + 1\right) d(u,v) \\ &= t\,d(u,v). \end{aligned}$$

Let's now show how you can efficiently create such a diagram.

k-Center Clustering

Given some metric space (P, d) & $k \in \mathbb{N}$, find some centers $C \subseteq P$ w/ $|C| = k$ minimizing

$$\max_{p \in P} \min_{c \in C} d(c,p).$$

We call this quantity the coverage radius.

This problem is NP-Hard, so we won't formally solve it, but we will do an approximation.

We iteratively create a 2-approximate k-center, that is the coverage radius w/in a factor of 2 of optimal.

Algorithm:

First, let's solve $k=1$. To solve this, just take any point $c_0 \in P$. Let $C = \{c_0\}$. This is a 2-approximate

i.e. $Opt = \{opt_0\}$

We prove this now. Let $Opt \subseteq P$ w/ $|Opt| = k = 1$ be the optimal solution. By definition

$$\text{radius}(Opt) = \max_{p \in P} \min_{c \in Opt} d(c,p).$$

We show $\text{radius}(C) \le 2\,\text{radius}(Opt)$

Consider some $c \in C$ & $p \in P$. By the triangle inequality we know

$$d(c,p) \le d(c, opt_0) + d(opt_0, p) \le \text{radius}(Opt) + \text{radius}(Opt) = 2\,\text{radius}(Opt).$$

(triangle inequality; property of Opt)

Now to find the next point, pick the farthest point $c_1 \in P$ from c_0.
So $C = \{c_0, c_1\}$.

Again, to prove it's 2-approximate let $opt = \{opt_0, opt_1\}$ be optimal, w/
$r_* = \text{radius}(opt)$
Consider the case where $c_0, c_1 \in \text{ball}(opt_0, r_*)$ (or opt_1). Consider some $c \in C$ & $p \in P$. Then
$$d(c,p) \le d(c_0, c_1) \le 2r_*$$
Now when c_0 & c_* are in separate balls then
$$d(p, c_i) \le 2r_{*0}$$

Generalizing this gives us a <u>greedy permutation</u> which approximates the k-cover for any metric & k.

<u>Def:</u>
P has doubling dimension d iff $\forall p \in P$ & $r \in \mathbb{R}$ $\exists C \subset P$ st
i) $|C| \le 2^d$
ii) $\text{ball}(p, r) \subseteq \bigcup_{c \in C} \text{ball}(c, r/2)$