CS 6301.008.18S Lecture—January 9, 2017

Main topics are #introduction, #administrivia, and #convex_hulls.

Introduction

- Hi, I'm Kyle!
- Welcome to CS/SE 6301.008.18S – Special Topics in Computer Science – Computational Geometry.
- Computational geometry emerged from the field of discrete algorithm design and analysis in the late 1970s.
- It involves designing algorithms and data structures for different geometric problems. Since it is a subfield of algorithms, I'll be emphasizing provably correct algorithms with low worst-case running times. Think 6363, but with more points and lines.
- So it is mostly a theory class, but I believe it should be very useful to those working in different application domains.
- For example, when trying to draw scenes in computer graphics, you need to know which objects are visible from certain pixels and which are obscured by other objects. One method of solving this problem is to perform *ray tracing*, figuring out which object is hit first if I shoot a ray from a pixel of my screen into the scene.
- In geographic information systems, we need to know what information should be displayed based on how zoomed in we are. And if somebody points at a location in a map, you need to figure out which city they're pointing at. Both problems require fast geometric data structures.
- In robotics, we need to understand how to move robots around while avoiding obstacles. A simple version of this problem is asking how to move a robot from point a in the plane to point b, while avoiding a number of polygonal obstacles.
- So while the class will mostly focus on the algorithms themselves, I hope you'll learn about some useful techniques or algorithmic problems you hadn't encountered before that can then be applied in your work.

Administrivia

- Before we get into any more detail, though, I'd like to talk about administrivia.
- Everything I'm about to say can be found on the course website: https://utdallas.edu/~kyle.fox/courses/cs6301.008.18s/
- The course has one official prerequisite: CS 5343–Algorithm Analysis and Data Structures which should teach you about all the fundamental non-geometric data structures we'll use in this class.

- That said, you'll probably have a much easier time of it if you've taken CS 6363—Design and Analysis of Computer Algorithms. I'll be using a few algorithms from that class like Dijkstra's shortest path algorithm as subroutines, and some algorithm design techniques like divide-and-conquer and dynamic programming will likely come up.
- I'll try to fill in any additional background you may have missed, but I can't always guess what you know. Please ask lots of questions!

- Your grade will be based on a weighted average of three things.
  1. 60% will come from homework assignments.
     - I'll release homework once every few weeks and make it due a week or two later.
     - You may work in formal groups of up to three people if you'd like, but it is not a requirement to group up. Each group should turn in **one** copy of the homework with all your names on it.
     - I'll probably do submissions on eLearning depending on how many feel comfortable typing or at least scanning their homework. [how many feel comfortable typing your homework?]
     - I won't be accepting late homework, so please be punctual.
  - The rest of the grade will come from a course project. This can be anything suitably project-like such as a survey, implementation for some application or performance testing, or even original research in computational geometry.
  2. Each *individual* student will turn in a one to two page project proposal saying what you'd like to do. These proposals are worth 10% of your grade. I'll spend a day or two summarizing various proposals to the class.
  3. Finally, you'll get to form groups of up to three people again for the final project, which means you don't have to go through with your specific proposal if you like somebody else's better. 30% of your grade will be a short paper of around 10 pages and a 20 minute presentation describing your progress.
     - For research proposals, I expect almost all reports to be short surveys and descriptions of a few failed attempts. But if you partially or fully solve your problem, then all the better!
- I'll add up all of these things and **then** heavily curve the grades. This is a special topics course, so you shouldn't fret. I expect that everybody who puts in a good faith effort to get a good grade.
- Let me reemphasize, **there are no set percentage targets you need to meet to get certain grades**.
- That said, this curving scheme means I can ask tough questions and grade the homework ruthlessly.
- I think fighting back against the homework is your best opportunity to learn, and me being ruthless with grading is your best way to get meaningful feedback on what you learned.

- So don't high percentages. Again, there's no set scale.
- Also, algorithms is a theory topic, so you need to justify your answers or your algorithms with an argument, some might say a proof, that they are correct. Try to be at least as rigorous as I am in lecture.
- As a rule of thumb, if you can't explain why your answer is correct, it probably isn't.

- I need to set office hours. I was thinking Tuesdays at 2pm. [how does that sound?]
- The required textbook is Computational Geometry—Algorithms and Applications by de Berg et al.
    - I admit it's not the greatest piece of writing, but it is the standard text for computational geometry classes.
    - I won't be asking homework problems directly from the book, and I'll post my lecture scripts online, so if you *really* don't want to buy it you don't have to.
    - But, it's probably a good book to have on your shelf even after class is over.
- There's also a link to some lecture notes by David Mount on the website. I'll let you know what chapters from the books and lecture notes are relevant to whatever we're covering. And by the end of the semester, I'll probably be completely off script from both texts.
- Oh, and a warning about my notes: They should be fairly thorough, but I probably wrote them the day before class in Markdown. They aren't pretty, and they may have bugs.

- Alright, final bit of administrivia. I've posted an "assignment" to eLearning to fill out these prerequisite forms saying you've taken 5343. Taking 6363 or harder **instead** is more than fine. Please give me the filled forms after class or Thursday and I'll mark the assignment completed.
- And with that, let's actually talk some about computational geometry!

Convex Hull Definitions

- I want to start with the standard first problem in computational geometry courses: computing convex hulls in 2D.
- Here, we're given a finite set P of points in the plane, R^2. Let n := |P|.
- Informally, the convex hull is what you get if you wrap a big rubber band around all the points and let it snap down around them.
- It provides a simple summary of your point set.
    - For example, I can tell how spread out a point set is in any direction by looking at how spread out the hull is in that direction.
    - Convex hulls can also be used to approximate objects other than point sets. I could take the convex hull of a polygon's vertices for example. If I have some polygons representing objects floating through space, I can use their convex hulls to simplify

collision detections with a small loss in accuracy.

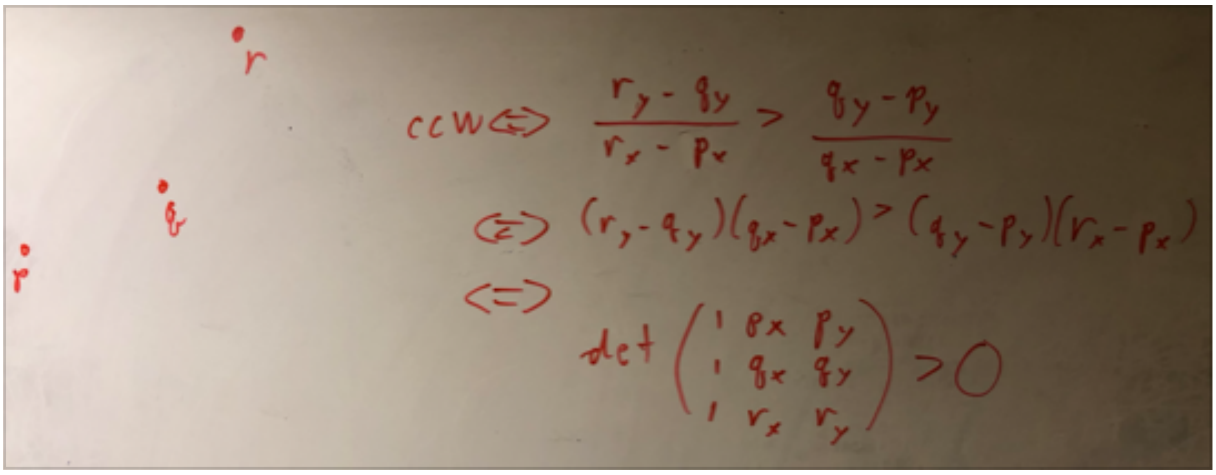
- We want to compute the convex hull of P, but we also want guaranteed correct algorithms, so we should probably be a bit more formal.
- In d-dimensional space, a set K subseteq R^d is *convex* if for any pair of points p,q in K, segment ~~pq~~ is entirely contained in K.
- Prop: the intersection of two convex sets is convex.
- The *convex hull* of P is the intersection of all convex sets containing P. The previous statement implies the convex hull is just the smallest of all convex sets.
- Since P is finite, that means the convex hull is actually a polygon, and the vertices of the polygon all come from P.
- What we'll compute is a counterclockwise list of vertices lying along the convex hull.
- But to make this feasible, I want to do a quick aside on how we make certain assumptions in computational geometry to keep things clean avoid getting bogged down in details.

Assumptions

- Computational geometry is an offshoot of discrete algorithms, so we make some assumptions to keep things clean.
- First, we'll usually assume that our input is a collection of real numbers instead of floating point numbers. That way we can get to the algorithmic meat of the problem without worrying about computation details. We can usually modify algorithms designed for real number inputs to work nicely with floating point inputs as well. I won't get into it today, but Chapter 1 of the book has some nice asides on the topic.
- The other big assumption we'll make is that our inputs often lie in what's called *general position*. Informally, this means we have no degeneracies in our input that might force us to deal with tons of annoying special cases. When the input is a set of points like P, we'll assume:
  - No two points share the same x or y coordinate.
  - No three points lie on the same line.
- Like real numbers vs. floating points, you can usually design an algorithm assuming general position and then do some simple modifications to make it work with all inputs, but only *after* we design the initial algorithm.
- Again, we design the algorithm assuming these nice things *first*. Then, if you need to implement it, you can fill in the extra details.

- Finally, we'll often assume that we can do simple things involving a constant number of objects in constant time.
- For example, the algorithm I'm going to talk about today assumes we can take three points <p, q, r> and determine if they do a "left-hand turn" or a "right-hand turn" in constant time.

- In this case, it's actually pretty simple to show you how to do this, so I'll go ahead and do it now. And no, it does not involve trig.
- Let's say our three points p, q, and r go in that order from left to right.
- We can determine if the do a left-hand turn by comparing slopes.



$$\text{ccw} \iff \frac{r_y - q_y}{r_x - p_x} > \frac{q_y - p_y}{q_x - p_x}$$

$$\iff (r_y - q_y)(q_x - p_x) > (q_y - p_y)(r_x - p_x)$$

$$\iff \det\begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix} > 0$$

- And I'm not going to go through all the cases, but it turns out this one test works no matter how p, q, and r are arranged.
  - If the determine is greater than 0, then left-hand turn.
  - If less than 0, then right-hand turn.
  - If equal to 0, then they're on a line (and not in general position).

Algorithm for Convex Hulls

- So with the time remaining, I'd like to discuss actually computing convex hulls.
- I'll present a modification by Andrew ('79) of the well known Graham's scan ('72) algorithm that runs in O(n log n) time.
- This algorithm uses an idea called *incremental construction* that will come up many times over the course of the semester.
  - We'll add points to a collection one-by-one, maintaining a solution for just the points in our collection.
- Usually, we like to work with the points in some convenient order, so we'll start by sorting them from left to right in O(n log n) time. Let p_1, …, p_n be the points in this order.
- It's also convenient to find convex hull vertices in order form left to right.
- Now, you can't trace around the entire hull going left to right the whole time, but if you split the hull into an upper hull and lower hull, things work fine.
- We'll focus on finding the upper hull. The algorithm for the lower hull is symmetric.

- So like I said, we'll add points to our collection one-by-one, also maintaining the upper hull of the points added so far. In other words, after adding point p_i to our collection, we should have an upper hull for points P_i = {p_1, …, p_i}.
- There's a few ways we could implement this, but as suggested by Mount's notes, we'll store

the upper hull computed so far in a stack S where S[top] refers to the rightmost point in the upper hull, S[top - 1] refers to the next point to the left and so on.

- Now, remember, the points p_1, p_2, … are sorted in left-to-right order. There are two points guaranteed to be in the upper hull for P_i = {p_1, …, p_i}.
- Right, so p_i *always* gets added as the rightmost point of the the upper hull. But do other point get to stay?

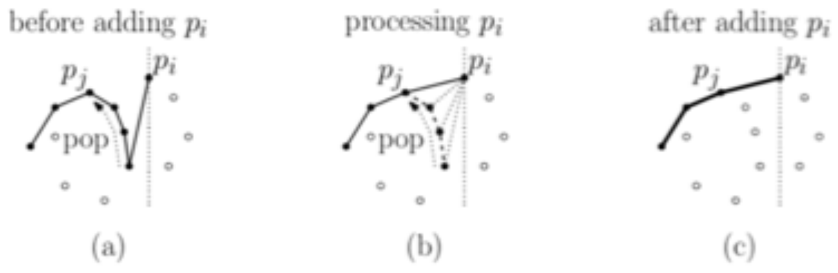
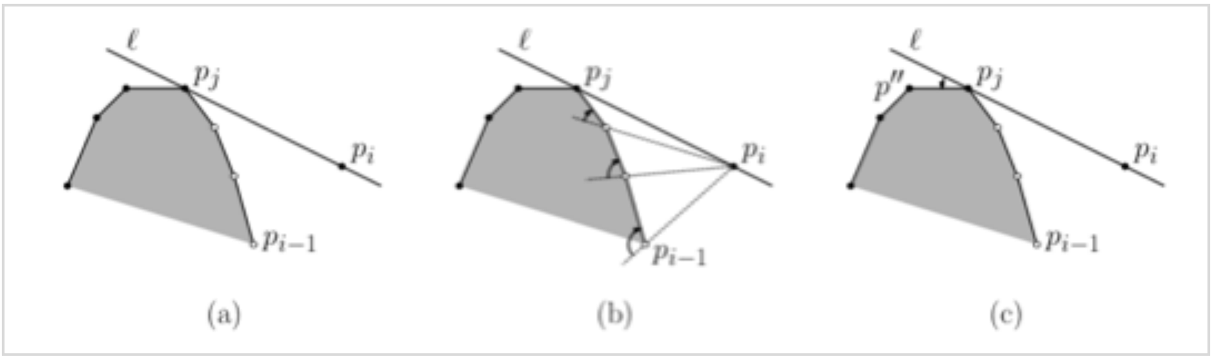


Fig. 14: Graham's scan.

- Well, if <p_i, S[top], S[top - 1]> form a right-hand turn, then S[top] has to go! The upper hull goes over S[top], so we should pop it from the stack. Then we repeat the test, popping, popping, popping, until finally we have only p_1 remaining in the stack or <p_i, S[top], S[top - 1]> forms a left-hand turn.
- Then we can add p_i to the upper hull.
- UpperHull(P):

```
• Sort points by x-coordinate increasing to form <p_1, …, p_n>
• push p_1 and p_2 into S
• for i ← 3 to n
    • while (|S| ≥ 2 and <p_i, S[top], S[top - 1]> make a right-hand turn
        • pop from S
    • push p_i into S
```

- So we should prove that this actually works. We'll prove the following claim inductively:
- Claim: After inserting p_i, the vertices of S from top to bottom form the upper hull of P_i = {p_1, …, p_i}.
- Proof:
  - Claim clearly true after inserting p_2 since the upper hull uses both p_1 and p_2.
  - So, we add p_i, which must be in the upper hull of P_i since it's furthest to the right.
  - Let p_j be the next point to the left on the upper hull of P_i.



  - No point z of P_i lies *above* the line through p_j and p_i; segments zp_i and z_pj contain points above the upper hull.
  - But that mean p_j is on the upper hull of P_{i-1}; otherwise there would be points above it. In particular, p_j is in S when we enter the for loop the ith time by induction.
  - Now for each p_k where j < k < i, we have the turn from p_i, p_k, and p_k's predecessor on the hull of P_{j-1} forms a right-hand turn.
  - Each p_k lies strictly below the line, so it is correct to pop it off the stack. And the turn at p_j is left-hand, so it doesn't get popped. Good.
  - Finally, we push p_i into S giving us the final correct hull.
- The running time for computing the upper hull is relatively easy to figure out.
- Sorting takes O(n log n) time.
- Let d_i be the number of pops when inserting p_i.
- We do one orientation test per pop and one more before inserting p_i, so the time adding p_i is O(d_i + 1).
- In total, we spend time proportional to sum_{i=1}^n (d_i + 1) = n + sum_{i=1}^n d_i building the upper hull after sorting.
- But each node is popped at most once, so sum_{i=1}^n ≤ n. The total time building the upper hull is O(n) in addition to the time for sorting or O(n log n) total.

- On Thursday, we'll spend a little more time working with convex hulls and see how sometime geometric algorithms run much faster when their outputs are small.