CS 6301.002.20S Lecture 1—January 14, 2020

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

Introduction

- Hi, I'm Kyle!
- Welcome to CS/SE 6301.002.20S — Special Topics in Computer Science — Computational Geometry.
- Computational geometry emerged from the field of discrete algorithm design and analysis starting 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'll spend a lot of time focusing on material useful in relevant application areas.
- In fact, having this knowledge may help you differentiate yourself when looking for jobs!
- 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://personal.utdallas.edu/~kyle.fox/courses/cs6301.002.20s/
- The course has one official prerequisite: CS 5343—Algorithm Analysis and Data Structures. This course should teach you about all the fundamental non-geometric data structures

we'll use in this class and maybe some basic algorithms we'll use as subroutines.

- 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. 50% will come from homework assignments.
     - I'll release homework once every few weeks and make it due about two weeks 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 on eLearning. I'll give everybody in the group the same grade.
     - I'll take late homework, but you'll lose 10% of the maximum points if you're 0 to 24 hours late and 20% of the maximum points if you're 24 to 48 hours late. After that, I'll stop accepting the assignment for credit.
  - 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 put your proposals up on eLearning so they'll be visible to the rest of the class but not the greater internet.
  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. 40% of your grade will be a short paper of around 10 pages. Depending on how many people stick around for the first week, I may also ask your group to do a 20 minute presentation describing your progress. Right now I'm getting a bit worried we won't have time for presentations, though. I'll let you know if we're doing presentations as soon as possible.
     - 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.
- 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 so you're not

surprised, I like to grade 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 expect 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 after appealing to things in class and your own logic, it probably isn't a correct algorithm.
- The website contains a whole page devoted to what you should and shouldn't do when answering homework questions.

- I am tentatively planning to do office hours Mondays from 10am to 11am and Tuesdays from 2 to 3pm in my upstairs personal office. I can also set additional meetings by appointment. If those regular times don't work for too many of you, they can be changed.
- The required textbook is Computational Geometry—Algorithms and Applications by de Berg et al.
    - I find it a bit long-winded and loose myself, 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 is a good book to have on your shelf even after class is over. I refer to it occasionally for my own research.
- There's also a link to some lecture notes by David Mount on the website. My lectures will largely follow his presentation. 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 since we'll be focusing on less classical topics.
- 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 either already posted an "assignment" to eLearning or will shortly 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

- If you've taken 6363, especially with me, you probably know I like to emphasize techniques over specific problems.
- In this class, you'll see more of the opposite, with specific problems motivating techniques.
- In particular, I want to start with the standard first problem in any computational geometry courses: computing convex hulls in 2D. This problem has a nice combination of being easy to describe, practical motivation, and having *several* algorithms, all of which demonstrate different techniques commonly used in computational geometry.
- We'll begin with a few definitions suitable for any d-dimensional Euclidean space R^d. Some of these definitions will prove useful later.
- A set K subseteq R^d is *convex* if for any pair of points p and q in K, the line segment pq is entirely in K. So points within a circular disk or polygon are convex. So are line (segments), rays, or *halfspaces*, all the points lying to one side of a line in the plane or to one side of a hyperplane in higher dimensions.
- The real formal definitions are a bit messy and belong more to a topology course, so we'll say a set is *open* if it does not include its boundary and is *closed* otherwise.
- A set is *bounded* if it can be enclosed in a sphere of fixed radius. Otherwise, it is *unbounded*. So line segments and circular disks are bounded, but lines, rays, and halfspaces are not.
- A *convex body* is a closed, bounded convex set. The convex hull will be one of these.
- We'll rely on the follow fact: Let K be a convex set in R^2. For every point p on the boundary of k, there is at least one line ell such that K lies entirely in one of the two halfspaces above or below ell. For R^d, consider ell to be a hyperplane instead.
- Finally, given a set of points P, the convex hull of P, denoted conv(P) is the intersection of all convex sets containing P. Equivalently, it is the smallest convex body containing P.
- For today and Thursday, we're given a finite set P of points in the plane, R^2. Let n := |P|. Set conv(P) is a polygon whose vertices come from a subset of P.
- Informally, the convex hull is what you get if you wrap a big rubber band around all the points of P 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 our algorithm to provide a good description of conv(P), so we'll have it compute a counterclockwise list of vertices lying along the convex hull.

Assumptions

- Computational geometry is an offshoot of discrete algorithms, so we make a couple assumptions to keep things clean while designing our algorithm.
- 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.
- After all, how can you handle the complicated case if you can't even handle the "easy" case first?

Algorithm for Convex Hulls

- Today, 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. After computing both, we can concatenate their lists in constant time to compute the whole convex hull.

- 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}. These are p_1 and p_i.
- So, p_i *always* gets added as the rightmost point of the the upper hull. But do other points get to stay?

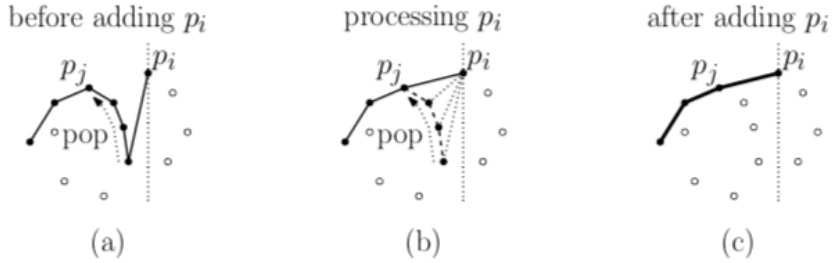
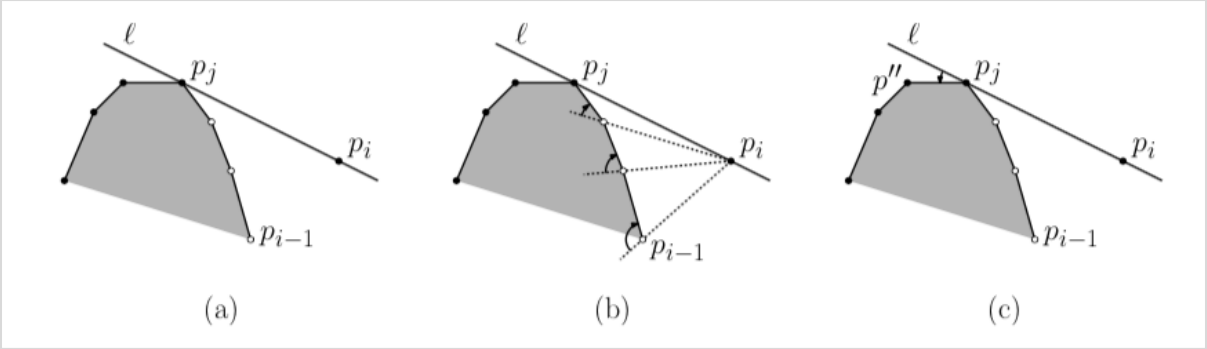


Fig. 14: Graham's scan.

- Consider the ordered triple, <p_i, S[top], S[top - 1]>. I'll prove the following: If we do a right-hand turn following these points, 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 and we're done with that iteration.
- 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 using induction on i:
- Claim: After inserting p_i, the vertices of S from top to bottom form the upper hull of P_i = {p_1, …, p_i} going right-to-left.
- Proof:
  - The claim is clearly true before the for loop, because the upper hull of p_1 and p_2 is the line segment p_1 p_2.

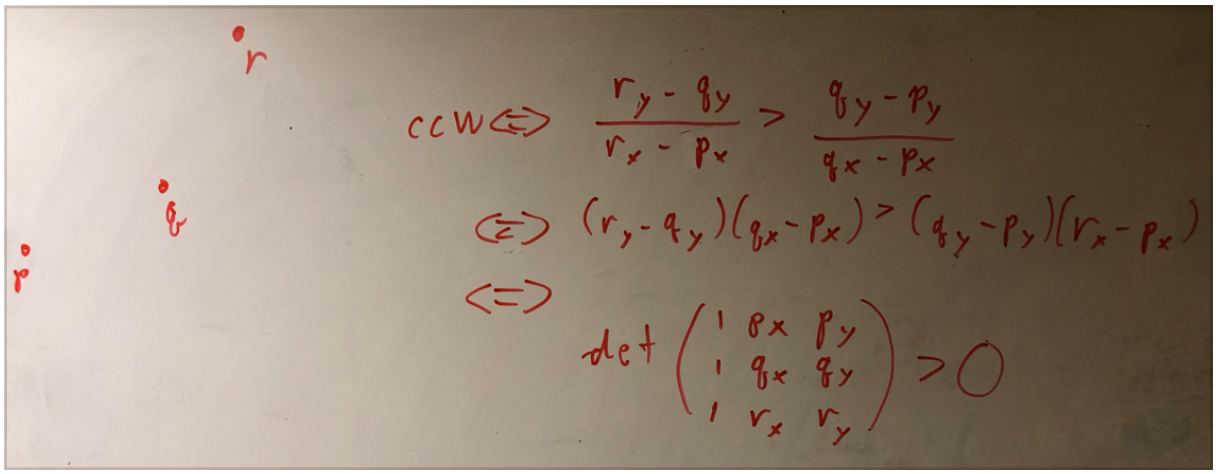
- Now, consider when we add p_i to our collection to create P_i. p_i must be in the upper hull of P_i since it's furthest to the right.
- Let p_j be the next point to the left on the upper hull of P_i.



- 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 means p_j is on the upper hull of P_{i-1}; otherwise there would be points above p_j. In particular, the induction hypothesis guarantees p_j is in S when we enter the ith iteration of the for loop.
- Now for each p_k where j < k < i, we have that 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.

Testing Turns and Final Details

- All that remains for designing the algorithm is to find a fast way of determining if a triple makes a right hand turn or left hand turn. It turns out we don't need trig for this!
- Let say we have three points <p, q, r> going from left to right and we want to know if <p, q, r> is a left-hand turn.
- We can determine if they do a left-hand turn by comparing slopes. Let o_x and o_y be the x and y coordinates of point o.



- I'm not going to go through all the other cases, but it turns out this one test using the

determinant 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).
- We've designed an algorithm for convex hull! Now for the running time.
- 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 sometimes geometric algorithms run much faster when their outputs are small.