[SQUEAKING] [RUSTLING] [CLICKING]

JASON KU: Hi, everybody. Welcome to the last lecture of 6.006. Last lecture, we talked about summing up this class and talking about future courses in the department that use this material. Just as a pointer to some of those classes, I have a little slide here I didn't get to at the last lecture, talking about what I was talking about at the end of the last lecture about different models-- different specialized classes on different aspects of 006 material-- for example, more graph stuff, different models of computation, randomness, complexity.

All of these things have their own specialized classes in the department, as well as a lot of applications for this material in subjects like biology, cryptography, and in particular, for your instructors, the realm of graphics and geometry. All of your instructors this term happened to be geometers and be interested in geometry-related problems. Me in particular, I didn't start out in computer science. I started out in mechanical engineering. And the thing that was my passion coming into MIT was origami. Here's a couple of pieces that I designed-- origami pieces, one square sheet of paper without cutting. Here's a lobster, and here's a copyrighted dinosaur from a particular movie of the year that I designed it.

When I was young, in high school, I started designing my own origami models. And what I didn't realize was, the procedures that I went about designing these models was actually algorithms. And I just didn't have the mathematical language to understand exactly what I was doing, but I could gain some intuition as an origami artist and design these things by using some of those algorithmic techniques.

It wasn't until grad school, as a mechanical engineer, that I started talking with our other instructor here, Professor Demaine, about using algorithms and computer science to design not just origami, which we both do, but also folded structures that can be used for mechanical applications like space flight, deployable bridges in times when you can't-- you need a temporary bridge or shelter or something like that. Deployable structures where you might need to make folded structures-- transformable structures that can have different applications for different purposes-- need to reconfigure.

The dream being that, we have these powerful devices in our pockets right now-- cell phones-- which are really powerful because we can reconfigure the bits in them to make software of all different kinds, right? There's an exponential number of different programs that we can write. And that's part of why you're here, is to write the next best one. Right? So that's how to make kind of a universal device at the electronic level. What if we could do that from a material standpoint?

What if I could reprogram the matter in my phone so that, not only could I reprogram the app that's on your phone, but instead of having, say, the iPhone 10 or whatever that you have, and you want to go by the iPhone 11, instead, you download a software app that then reconfigures the matter in your phone-- it folds or reconfigures into the next generation iPhone. You don't have to throw away that old one. You can essentially recycle the material that you have to potentially save material, save cost, and be better for the environment, potentially.

So I started moving into computer science because I found that it was a really good way to model the world and solve some really interesting problems about folding that I really enjoyed. The three of us today are going to spend some time talking a little bit about how we can use algorithms-- 6.006 material and beyond-- in our own research. And we're going to start off with Professor Demaine, and then Professor Solomon.

ERIK DEMAINE: Thanks. So let me just jump in here to computational origami and geometric folding algorithms, sort of a broader umbrella for folding-related things, which is encapsulated by this class, 6.849, which is happening next fall. So you should all take it. 006 should be a reasonable background. And in general, we're interested in two kinds of problems. One-- the big one is origami design, or in general, folding design, where you have some specifications of what you would like to build.

In this case I wanted to make a logo for 6.849. And I imagined extruding that text into third dimension. And then I wanted an algorithm to tell me how to fold that structure. And so there is an algorithm, which I'll talk about in a moment, that gives you a crease pattern. And then, currently, you fold it by hand. The dream is, we'll eventually have folding machines that do it all for us. And so that's the origami design, where you go from the target shape back to the crease pattern.

The reverse direction is sort of foldability. If I gave you a structure like this and I wanted to know, does it fold? That's the problem we call foldability-- in general, class of problems. And sadly, most of those problems are NP-hard. Jason and I proved that foldability is hard for a general-- given a crease pattern like that, telling you whether folds into anything, it turns out to be NP-hard. So that's bad news. So we focus a lot on the design problem, because that actually tends to be easier. We can solve it with algorithms like that one you're seeing.

A long time ago, we proved that you can fold everything. If I give you a square piece of paper and you take any polygon you want to make-- or maybe the paper's white on one side, black on the other, you want to fold some two-color pattern, like a zebra, or in general, some three-dimensional surface, like these guys, there is a way to fold it from a large enough square of paper. And it's actually really easy to prove that with an algorithm. I have the sketch of the two pages of proof that we go over in 6.849, but I'll just hand-wave a little bit. If you take a piece of paper, like my lecture notes here, the first thing you do is fold it down into a very long, narrow strip-- much longer and narrower than this one-- wasting most of the material. And then you take your strip, and you just figure out how to turn it in some general way, and then you just sort of zigzag back and forth along the surface.

So it's very cool in that you can prove with an algorithm, and in a very short amount of time, to someone you can actually fold everything. Of course, it's a terrible folding, because in the very first step, we throw away all but epsilon of the material. But it's a starting point. That was back in the '90s-- late '90s-- one of the first results in computational origami. And in modern times, we look for better algorithms that are more efficient, that try to minimize the scale factor from, how big of a piece of paper do I start from to, how big of a model do I get?

And one of the cool ways these days, which was invented by Tomohiro Tachi and then analyzed by the two of us-- it's called Origamizer. It's free software. You take a 3D model and you can-- it makes it into a pattern that you fold from a square. In this case, it uses 22% of the area, which is pretty good-- similar to these guys in terms of efficiency. But very, very different kind of folding than what you would get from more traditional origami design, which uses different algorithms, which I'm not going to talk about. But you should take the class. Jason gives a lecture in the class, so you can learn from him.

But the vision is, we can take any sheet of material that can hold a crease, like this sheet of steel that Tomohiro is folding. It was cut by a big laser cutter at MIT. And this is him in this Data Center several years ago, folding it into a steel bunny. And so this is a totally new way to manufacture 3D objects. And you can make particularly interesting objects that either collapse flat for transportation or transform, like Jason was talking about. But I'm just giving you a flavor. I think the first paper we wrote together was on maze folding. So this is an example of folding a maze from a rectangle of paper. And you can all try this out. You just google for our Maze Folder. You can generate a random maze. And this 3D structure can be folded from this crease pattern. That's a really hard one, so maybe try something smaller.

You can also write your favorite message and fold this maze-- extruded graph-- from this crease pattern. Might want to start with something smaller, but that's the general idea. And it's actually quite easy to prove this algorithmically, if you have a really good origamist like Jason on your team. What you do is design how to fold each type of vertex. This is just a graph on a grid. There are some constant number of different ways that each vertex could look. It could be degree 4. It could be degree 3, as a T. It could be degree 2, either a turn or a straight. And you design little gadgets, little crease patterns, that fold into each of those little structures.

And if you can do it in a way that these boundaries are compatible, then to fold the whole thing, you just sort of gluon together those crease patterns. And that's how that software works. This was particularly interesting, because you can fold an arbitrarily complicated graph-- arbitrarily complicated maze, n by n, with a constant scale factor. As long as the height that you're extruding that maze is constant, then this is one family of shapes we know how to fold really well. In general, we're trying to understand, what makes this lobster a nice shape in that it can be represented with a not-too-large piece of paper. And we don't have general answers to that problem.

I think that was a whirlwind tour of computational origami. I also play a lot in algorithmic sculpture. One of the leading edges in origami and origami math is understanding how curved creases work. And one of our favorite models is this one, where you fold concentric circles alternating mountain and valley, cut a circular hole out, and it folds into this kind of Pringle shape as a nice physics equilibrium thing. And then you can turn it into fun sculptures like this.

These are done with my dad, Martin Demaine, who's also here at MIT, or this guy. This paper has been printed with a pattern according to getting burned by glass. And then it gets folded and then put inside glass, also. Made here at MIT. We use sculpture to try to explore and understand intuitively how curved creases work, and then we get better and better understanding of the mathematics of even-- we don't even know whether this surface exists, whether it's possible to fold in this way, although getting close to proving it.

That was sort of in the top level of this hierarchy. Computational geometry is a bigger umbrella, which is represented by another class, 6.850, that's being taught this term. And then I talked about geometric folding within that branch. Let me briefly tell you about another world of geometry-- very different in terms of model of computation. Oh, I jumped ahead a little bit. Rewind. Let me show you one more fun demo, which-- if I find my scissors. If I take a rectangle of paper, and I fold it flat and make one straight cut, what shapes can I get? It's called the folding cut problem. It's hundreds of years old.

Here, for example, I get a swan. Here, I get-- one straight cut. I unfold and get angelfish. Tough audience today. I've got to keep going. You've seen all of these before. This is this one is a particularly difficult one to fold-- to only fold. And to cut, yeah. OK. That works well. This is the MIT logo. Ooh, ah.

AUDIENCE: Ooh, aah. MIT, yeah!

ERIK DEMAINE: Yeah. Go, MIT. All right. That's actually the first problem I worked on in computational origami. It's a lot of fun. And there's a really interesting algorithm here, also, for computing the crease pattern, how to fold your piece of paper to align-- in fact, any graph you draw on a piece of paper, you can align all of those edges and nothing else. So you cut along the line and you get exactly what you want. Cool. All right.

Now, I want to talk about something completely different, which is self-assembly. A fun thing you can do with DNA, which we all have. Just pick out some cool DNA strands and design them in a clever way so they fit together to form a kind of square with dangling ends, which I'll call glues and each of those dangling ends can have a very particular pattern, and only identical or complementary patterns will attach to each other. And so you can use this to design your own self-assembling system, like biology does, but engineered, for example, to build a computer. This is an example of taking a bunch of these square tiles and building a binary counter. This thing is roughly counting in binary along the diagonal. It's a little skewed, so it's hard to see.

But the general model is, you have squares-- this is sort of the computational model-- with four different glues. And you can build any square you want, but you don't have very many of these different glues, ideally. And then, if you have two tiles with complementary glues, they will want to match together. But it depends how strong this glue is, how much affinity there is for how long those DNA dangling ends are, and also, the temperature of your system. If you have really high temperature, nothing. Will stick together low temperature, things will stick together even if they're not supposed to.

If you tune your system really well, you can design a system so that maybe these guys-- these glues are really strong. And so let's, I don't know, write "E" here-- Erik. And so these tiles will always glue together, but only when all three of these are glued together can this tile-- which has C complement and F complement. Then it will, if you set the temperatures just right, only because both of these edges match will this dial be able to come in. And that's the basis for building that binary counter.

This is a very different model of computation from what we're used to in this class, where you think of instructions, and they run one at a time. Here the, model of computation is geometric. It's these squares that are just floating around and gluing together. And so your program, at any moment, is some conglomerate of squares. I just wanted to mention it because it's a really fun model. You can prove cool things in this model, like how to build any shape by a sequence of pores mixing between tiles that you can execute in parallel. And so it only takes log and time of parallel steps, a linear number of different mix operations, to make an arbitrary shape-- even using a constant number of different glues, which is cool, and maybe practical.

You can also use it to build a replicator, where you're given an object like this that you don't know the shape of-- like, we don't know whether this exists, and we can't model it mathematically very well, and you stick it in a vat, and all of these tiles would attach and basically build a mold, and then start photocopying, in 3D, that mold. And you can build that with a system with only two steps, I believe, and a constant number of tile types. And it does all of that, in this model, in constant time. In reality, you would have to feed this machine and wait for it to print out all of these things, and these experiments take hours, if not days, to run. But in theory, it's really cool. And you get some really fun models and very general results. You can also use it to build a miniaturizer or a magnifier and other fun stuff.

That was a brief tour of computational geometry. I work mostly in four different areas of algorithms-- geometry, data structures, graph algorithms, and what I call recreational algorithms. I think I made up that term. And let's go into data structures, which is represented by this class, 6.851. All of the classes I mentioned have online video lectures, especially for those watching at home on OpenCourseWare. Most of these classes are on OpenCourseWare, and if not, they're on my webpage. 6.851, Advanced Data Structures, is an extension of the sorts of data structures you've seen here, in 006 and the ones you will see in 6.046.

I thought I would give you a flavor of one such result, which is a problem we've seen in this class done better. Suppose you want to store a dynamic ordered set. This is the set interface. Dynamic in the sense that I have insert and delete, and ordered in the sense that I want to support find-next and find-previous. Exactly which subset of the set interface you choose influences what data structure you've seen. We've seen, for dynamic sets, you want to use hashing. If you don't care about find-next, if you just care about find, then hashing is great-- constant expected. You can prove stronger things about hashing. And we do in that class.

But if you want dynamic and ordered, you cannot do constant time per operation. You can prove that, which is cool. What data structure have we seen that solves this problem pretty well? Set AVL trees, which solve everything in log n. So log n is one competitor. Yeah. I'm interested in the word RAM model, which is the only model we've seen in this class. This happens to work in a stronger model. And we can do better than log n in the following-- it will take me a while before I get better, but here's, at least, a different bound we can get-- log w. This is via a structure called van Emde Boas, who is a person. AVL is two people. van Emde Boas, I've actually met.

Log w-- remember, w is our word size. So this is a bit of a weird running time. It's great if w is log n, then this is log log n. And we know w is at least log n, but it could be bigger. We don't really have a sense of how big w could get. Maybe it's even n. Maybe it's big-- and then these are the same. Maybe it's bigger than n, and then this is maybe worse. But for most ws, this is actually pretty good-- and indeed, optimal. But it's not strictly better, in any sense, yet. On the other hand, there's another data structure which runs in log n divided by log w. This is called fusion trees. This was invented around the time that cold fusion was in the news, and so they wanted data structures to represent.

We can achieve this bound or we can achieve this bound. And this bound is good is if w is large. This band as good if w is small. You can always take the min of the two, whatever is better. And in particular, the min of those two things is at most-- I think it's square root log n over log log n. If you want to bound just in terms of n, then the crossover point between these two is this place. And so you're always, at most, this, which is quite a bit better than the log n of AVL. We've got a square root and we've got a slight thing in the denominator. Pretty tiny.

But the big thing is the square root. And that's kind of cool. And it turns out, that's pretty much optimal. In terms of an n bound, this is optimal. The min of these two, in general, is roughly optimal up to log log terms. For fun, I threw up the actual formula for the right-bound, which is tight up to constant factors of matching upper and lower bounds, which we talk about. It's min of three things-- four things, including log of w over a divided by log of log w over a log of log n over a. That's the last term that I just read. This was messy. Surprisingly, that is the right answer for this very particular problem-- a very natural problem.

AUDIENCE: What is a?

ERIK DEMAINE: A is the log of the space you're using. So it's the address size. Good question. If you throw it-- so it depends. If you have a polynomial space data structure, then basically, these are optimal. And this is generalizing to beyond that. Maybe you have a little bit more than polynomial space. Cool. So that's data structures. I'm going to jump ahead to graph algorithms, which, if you want to take this class, I recommend a time travel device. Go back to fall 2011. It may never get taught again. But it has video, so you can watch-- instead of time traveling, if you don't want to watch it live, you can just watch the recorded version. It was taught by a bunch of postdocs that were here, and a bit myself.

What I like to do with graphs is the world of planar graphs, or near-planar graphs. We've talked a lot about, this class, algorithms that work for arbitrary graphs. And the algorithms we've seen in this class are pretty much the best we know for a lot of problems for arbitrary graphs. But if your graph has some structure, like it's a road network and there aren't too many overpasses, you can usually draw these graphs in the plane without crossings. That's the meaning of planar. Maybe not exactly. Maybe just a few crossings. There's a generalization of this, which I won't get into.

But let's just think about planar graphs. Planar graphs have some nice features, like they always have a linear number of edges. They're always sparse. So you can immediately plug that into our existing bounds. But even so, Dijkstra, in such a graph, would take v log v time. For planar graphs, you can do the equivalent of Dijkstra, meaning, I can compute single-source shortest paths with negative edge weights in linear time. No log. Not that impressive, but remove a log.

More impressive is, we can do the equivalent of Bellman-Ford, which is a single-source shortest paths with arbitrary edge weights in a planar graph in some time-- almost linear time. The log squared v over log log v. So there's a couple of factors here-- but for almost linear time, whereas Bellman-Ford would take v squared time. So this is a huge improvement over what we've seen in the class. These are quite complicated algorithms, but they're covered in that class, if you're interested in them.

Then the area I work in a lot is approximation algorithms for planar graphs. And let me just give you a fun flavor using something we know, which is breadth-first search. Breath-first search you can think of as building sort of rings around a single root node. And there's this general approach-- this was introduced by Baker in 1994, we've used for lots of different problems. We want to solve some NP-hard problem on a graph.

So just run breadth-first search from an arbitrary vertex and decompose your graph into these layers. You could number them-- 0, 1, 2, 3. These are levels. And let's just, like, delete some of those layers. Let's say, let's delete every fourth layer. So maybe I delete this one. I delete all of the vertices in that layer. And then I delete all of the things in layer 8, and layer 12, and so on. Guessing-- I don't know which one to start with, but from-- I'll just try them all. And then I delete every fourth layer after that. So I've deleted, on average, about a quarter of the graph.

And it turns out, for a lot of problems that you care about, like choosing where to place fire stations in this graph to minimize travel time for if there's a fire somewhere in the graph-- this happens, you know? Fires and graphs. Then this will only hurt your solution by, like, a factor of 1 plus a quarter. So you will get a solution that's within 25% of the optimal, for a lot of problems. And that works for any value 4. So I could do it for 10, and then I would get within 10% of the optimal solution.

OK, but how do I actually solve the problem once I delete every fourth layer? Well, then your graph has this extra special structure, which is a constant number of layers, let's say. A constant number of breadth-first search layers. If you just look at this portion, this connected component, or this connected component in here, you can-- your graph is almost like a cycle. It's like four cycles stacked up together with some connections between them. And it turns out, that's something you can solve with very fancy dynamic programming, like the stuff we've seen in this class, which focuses on just a single path or a single cycle.

If you just have a constant number of cycles, with more work, you can still do everything in polynomial time. This is a very general approach for getting arbitrarily good approximation algorithms. We call these 1 plus epsilon approximation for any epsilon. But the larger the epsilon, the more time you take. It's something like 2 to the order 1 over epsilon times polynomial n. So as long as epsilon is constant, this is polynomial time. This is called a PTAS. Anyway, that was graph algorithms.

Last topic is recreational algorithms, which is maybe best encompassed by this class. 6.892 is its latest name. It changes names every once in a while. And I mentioned it in the hardness complexity lecture, because this class is all about hardness proofs, analyzing fun games and puzzles. We saw the Tetris NP-hardness in that lecture. But you can also prove Super Mario Brothers is hard, or Portal is hard, or Mario Kart is hard, or The Witness, a modern video game, is hard. Or, one of our latest results is that Recurse-- that game in the top right-- is undecidable. There's no algorithm to play that game perfectly. And you can even download the level-- an example of the level and play it, if you dare. So that's a lot of-- we have a lot of fun in that world of hardness of different games and puzzles.

Where do I want to go next? OK. Next topic is balloon twisting. Totally different. This is recreational, but not about hardness. This is an octahedron twisted from one balloon. I made another one on a stick. Each of these is made for one balloon. What graphs can you make for one balloon? Well, you should read our paper. And you can characterize how many balloons you need to make each polyhedron. And some of these problems are NP-hard, and it's a lot of fun. Cool. I think that's the end of the slides.

The last thing I wanted to show you is a problem, a puzzle/magic trick-- it comes from the puzzle world-- called the picture hanging problem. So imagine you have a picture. You want to hang it on a wall. So you invested in some nice rope, and you hang it on a nail. If the nail falls out, the picture falls, and you're sad. So you invest in two nails, like I have here, and maybe you hang your picture on both those nails. Now, if one of the nails falls out, you still have a crookedly hung picture. If the other nail falls out, OK, it's gone. I want to hang a picture on two nails such that, if I remove either nail, the picture falls. So, Jason, pick a nail, left or right. Left, we remove. Make sure this doesn't fall off-- and, boom, the picture falls. Same wrapping. You can check-- you can rewind the video, make sure I did the same wrapping.

JASON KU: And take out the right.

ERIK DEMAINE: Then take out the right one. Good choice. Then, also, the picture falls. This is a classic puzzle, but you can generalize it. So let me do it for three nails, which is all I have here. This nail is sagging a little bit. y, x-- y inverse, x inverse. I think that's right. So this is one way to hang a picture on three nails such that, if I remove any of the nails, the picture falls. Justin, 1, 2, or 3? 2. OK. Yeah, I want to get out of the way and make sure I don't go over the edge here. Yeah. It's a lot easier to make this one work. But you can see, boom, picture falls there. And of course, imagine infinite gravity. And the picture falls. Ta-da!

You can generalize this to do essentially any-- what's called a monotone Boolean function-- on any set of nails. I mean, you can make any subset of the nails cause the picture to fall and any collection of subsets of nails to make it fall. Of course, if you remove more nails, it's still going to fall. That's the monotone sense. But otherwise, you can do an arbitrary pattern, which is fun. That's actually a result with Ron Rivest and a bunch of other people.

I think I'm approximately on time. So that was a quick tour. And there are obviously various classes here you can take. 6.892, the hardness class, was just offered last semester, so it probably won't be for a while. But all of these classes are online. Watch the videos, feel free to ask me questions. And now we have Justin. I left you space here for your outline. You don't have to, but I'll put your name.

JUSTIN SOLOMON: Thank you.

JASON KU: So Justin is also a geometer.

ERIK DEMAINE: Yeah, we've got a lot of geometry people in 006 this semester.

JUSTIN SOLOMON: Thank you. OK. I can't help but share that, on our instructor chat, Erik was texting that he was going to be-- he was somehow nervous that the applied guy would have all of the cool stuff to show off, and now I feel totally boring. [LAUGHING] Right. Yeah. We have three different geometry instructors in this class. In this class, I think we have many different flavors of geometry that are kind of represented in this room here, from mechanical engineering, to theory plus lots of other cool stuff, to whatever it is that I do.

I'm a professor, also, in CSAIL, and lead a group that studies slightly more applied geometry problems, in some sense, and in CSAIL, we kind of cross a lot of boundaries-- actually, closer to the math department than to the theory group and computer science, which I would argue is largely a historical artifact rather than anything interesting about computer science or math. Continuing in our whirlwind tour of interesting geometry classes here at MIT, I have some more fun things to add to the list. And we'll introduce some of the ideas in the next couple of slides here.

So normally, every fall, I teach 6.837, which is the introduction to computer graphics course. In fact, my background was working in an animation studio for a little bit of time, and got one movie credit out of it until they changed the standards for movie credits, and then that stopped happening. But in any event, if you watch-- what's that movie-- *Up,* with the old man. If you hit pause at just the right moment, you can find me right above the list of babies that were born during production. But in any event-- although computer graphics might not sound like an algorithmic discipline, I'll try to convince you guys that, in some sense, you could take just about anybody in our department, have them teach 6.006, and give a similar talk that, like, the material that you've encountered in this course is going to be relevant to your life.

The other course that I teach that might be of interest-- and actually, is a little more theoretically flavored-- that I teach is 6.838. So since Erik so kindly put my name on the board here, I guess I can draw The So the main object of interest in 6.838 is a particular thing called the simplicial complex. Usually, in 6.006, we spend a lot of time thinking about graphs. Let me draw you a graph. So I'm going to take a square and subdivide it. And now, let's say I put edges diagonally like that.

Now, in 6.006, this thing is just a bunch of nodes connected by edges. In fact, if I took this edge and I moved it down or something, it would be the same graph. But of course, in a lot of computer graphics applications, this thing also looks an awful lot like a square. And the reason is that, of course, the graph here contains triangles inside of it. And so for instance, maybe I think of my graph as a collection of vertices, a collection of edges. This is the sort of notation we've seen before. And then I add a third thing to my description, which is a set of triplets. That's a set of triangles here.

And we can take a lot of the algorithms that we've talked about in this class and extend it to this case. For example, here's a deceptively annoying one. Let's say that I want the shortest path between two vertices of my graph. We certainly have learned Dijkstra's algorithm. That's one technique to do that. And indeed, common practice in computer graphics, which is shameful, is on your triangle mesh, if you want the shortest path between two vertices, run Dijkstra's algorithm on the edges.

And let's see if that works really quick. Let's say that I want the shortest path between-- and, by the way, I'm going to assume the length of my edges are the lengths as I've drawn them on the board here. So it's like 1, 1, square root of 2. OK. So let's say I want the shortest path between the bottom left and the upper right. If I run Dijkstra's algorithm, we're in good shape, right? We get-- I'll let you do the computations at home. You'll get the path that is these two edges. But here's a really annoying thing. Let's say, instead, I wanted the shortest path from the upper left to the lower right. If I run Dijkstra's algorithm on this triangulated square, what's going to be the shortest path?

Yeah. In fact, there's a bunch of them. One of them might go all the way down, and then all the way to the right. What's the length of this path? 1, 2, 3, 4. Is that the length of the shortest path? Well, probably not. Well, we would like our shortest path to do something like that. But graphs don't know how to talk to triangles. And this is going to be a problem. In fact, it wasn't until fairly recently [INAUDIBLE] history terms that we were able to kind of work out the correct algorithm for the shortest path in a triangulated domain like this. And that's the runtime that we would expect. This is called MMP. I'm guessing Erik and Jason could do a better job describing it than I can.

But the basic idea of the MMP algorithm actually is a really-- happens to be a nice extension of the way that we taught Dijkstra's algorithm in 6.006, because they really do keep track of these level sets of the distance function. But now, the level sets have to-- oops-- have to window and edge like that when I compute shortest path, which is a giant headache. This is one of these algorithms that was known in theory about 10 years before anybody bothered to implement it in a way that they could convince themselves it ran in n log n time. And nowadays, there's a cottage industry in computer graphics research papers to implement this and then speed it up in different ways.

And sadly, the reality is that a different algorithm that we cover in 6.838 called fast marching-- which doesn't actually give you the shortest path, but some approximation thereof-- is faster, easier to use, and basically indistinguishable. In any event, in 6.838, we kind of have an interesting dual-mindset. We'll talk about a lot of algorithms that look like what we've done in whatever this class is-- 6.006. But at the same time, start to have a more geometric flavor, and we don't worry quite as much about [INAUDIBLE]. So in our computation model, oftentimes, we're kind of OK with real numbers, because that's not where the headache is.

And of course, when you write code in this class, you use double-precision floating-point. If you're more responsible, like in Jason's previous lecture, you should probably keep track of the number of operations to make sure that your error is counted. But I'm not sure that we really bother with that. In any event, this allows us to have two different mindsets. There's one mindset, which is discrete. There's another mindset, which is smooth. We think about understanding geometry, like these triangular domains, as an approximation of a smooth surface. And then we might want to do stuff like compute curvature and so on, which is really associated with computing derivatives, which of course, we'll have on these kinds of simplicial objects.

And that leads to this really fun area of math and computer science, whatever, called discrete differential geometry, which sounds like a contradiction in terms. And it's something that we covered in quite some detail in this course. So we build up, all of calculus, that the only calculations you're left to do are on the vertices and edges and triangles of a triangle mesh. And get pretty far, including some constructions of topology, like the Duran complex, and so on. I would argue, actually, if you take our course and then the differential geometry courses in that department, somehow, some of the indices and headaches that you often encounter in that world are much more concrete when you try to make them work on a mesh. In any event, I think I've already spent all of my time.

I can tell you a little bit about research in our group. I really lead kind of a weird, extremely [INAUDIBLE] group, where some of our students are essentially theory students-- touch your keyboard. I'm sorry. It was a reflex. But it was fast. All right. So we have some students whose background is in math, other ones that we're in autonomous driving industry and decided to come back and work in research. Because of that, we have this extremely broad set of research problems, everything from the sort of classic machine learning problems you might encounter in geometry world-- like if I have a self-driving car and I want to identify pedestrians and other cars on the road in an efficient and accurate fashion.

By the way, part of that is machine learning and deep whatever, but there's another part, which is algorithms. Because actually, what comes into your LiDAR scanner is on the order of [INAUDIBLE] with points and some minuscule fraction of time. And time complexity of your learning algorithm actually is really critical to get it right, and something that there are a lot of open problems right now, because it's really not compatible with the hardware architecture that these cars often use.

We also look at [INAUDIBLE] geometry problems, like if I give you data, can I find a geometric structure? So it's a classic example of natural language processing. When we use words like near and far, in terms of semantics and meaning, all the time. The question is, can we actually find an embedded of our word data into a geometric space to facilitate the statistical algorithms that we care about?

And of course, we apply geometry to lots of practical problems, everything from meshing and scientific computing, which I think is sort of a classic one-- in fact, I think we're the first group that sort of enumerated all of the cool things that may happen to decahedral meshes, which is this bottom figure here. I should show this to people. There's some fun things to look at there. To other practical problems, like taking-- Erik took a zebra and folded it. We can take a zebra and move its texture onto a cat or a pig-- or, actually, off the side of the screen. But if we don't move the paper, [INAUDIBLE] for the 3D scan of what it might [INAUDIBLE].

In any event, in my five minutes remaining here, I thought I would dig into a little bit of detail of two-- or maybe one application, depending on when Jason and Erik get bored. And essentially, my message for you guys is, of course, [INAUDIBLE]. I'm not really a central CS theory group member here at MIT. But unfortunately for you guys, 6.006 is unavoidable. Even if you want to go into deep learning, statistics, whatever-- data science-- you're going to encounter the material that you've seen in this course. And in fact, it's really the bread and butter of just about everything everybody does here in this Data Center.

So then, I'll give you two quick examples, one of which lifted from my teaching, one from my research. If you continue with me next fall, we'll teach 6.837, which is the Intro to Computer Graphics course. One thing that's always amazing to students is, these, algorithms that produce these really beautiful images, can fit in about 10, 20 lines of code. So really, this is totally facetious, because if you want those beautiful images and you use those 20 lines of code, you'll be waiting until the death of the universe to actually compute these things. But in any event, one nice one for rendering-- so drawing a bunch of shapes [INAUDIBLE], something called ray casting, or its better known cousin, ray tracing. Typically, the difference is whether your rays can bounce off of the surface and have a secondary thing. Right.

Here's the ray casting algorithm. Let's say I have a scene built out of spheres and cubes. I'm going to have a for loop over every pixel on the computer screen. For every pixel, I've got to discover what color that should be. So I shoot a ray from my eyeball through that pixel and find the first object that it runs into. It's not so hard to intersect a line of a sphere or a line of a cube. So what is that algorithm? I've given it to you on the screen here. Not too bad to think about. And I think you guys are all extremely well equipped to analyze the runtime of this, which is roughly the number of pixels times the number of objects. Because for every pixel, I've got to decide what object the ray out of my eyeball hits first. So I need a for loop over [INAUDIBLE]. Make sense? Cool.

So let's look at a basic rendering problem. In fact, Erik already secretly snuck this one in here. There's a very famous 3D model called the Stanford bunny. The Stanford bunny is actually a great example of a simplicial complex-- in fact, a manifold one, triangulated surface. Actually, I'm not sure it's manifold in its original form. But usually, it is. And this innocent-looking, extremely famous 3D model is actually quite pernicious. It's composed of 69,000 triangles. And if I wanted 1080p-- like a high def rendering of my triangle-- then, of course, there's two million pixels on the screen.

So if we look at our big O expression, roughly, our computation time scales like the product of those two big numbers. So just to render this ugly gray bunny takes me a pretty large amount of time. And in fact, the reality-- by the way, the bunny is this famous test case in computer graphics, so if you take my class, you'll be rendering buddies all day. The reality is, we don't want just grayed, flat-shaded bunnies. We want bunnies that are transparent, and reflecting stuff, and I shoot my bunny with a bullet and shatters into a million pieces, and all of these cool things. So of course that, ray casting algorithm, with each one of these new graphics features I add, only adds to the time complexity of the technique that I implement.

So pretty quickly-- and indeed, if you write your own ray tracer at home, which I strongly encourage you to do-- what you will discover is that a [INAUDIBLE] would be the technical phrase. What is our way out of this? Well, if you take it 837, you'll see that our way out of these problems, in graphics, is data structures and algorithms. It's completely unavoidable. For instance, obviously, we spent quite a bit of time in this course talking about AVL trees. In 837, we'll spend a big chunk of our tours talking about space partitioning trees. Here-- I actually forgot what kind of tree this is. I think it's a KD tree. Doesn't matter.

In any event, one thing I could do is take all of the triangles of my bunny, and I could put the entire bunny in a giant cube with the property that the cube is outside the bunny. Let's say I cast a ray and the ray doesn't touch the cube. Can the ray touch the bunny? No, right? It zings right past it. So suddenly, I just saved myself a lot of computation time, right? I don't have to iterate over all the triangles inside of the body to see whether they hit the ray or not, because I already convinced myself, by this conservative test, that I didn't hit even the bounding box of the whole bunny.

Well, that's sort of a nice order 1 speed-up. But depending on how big the bunny is relative to the size of my rendered image, that might not be a super useful efficiency test. But of course, what could I do? I could take the box containing the bunny, I could slice it in half, and now it's saying, does my ray hit the front or the back of the bunny? Or maybe both. That's where you've got to-- that's where things get gnarly. And so on. So now you have this nice recursive tree structure, where I keep taking the box containing my bunny and chopping it in half and placing-- in some sense, usually, the triangles-- maybe not the leaves of my tree, but [INAUDIBLE] that's probably good enough. You get a structure like what you see on the screen here.

And why should you do that? Well, remember, it takes pn time to render my image of my bunny normally. Well, now, the picture is actually misleadingly suggestive. But you might think that, maybe, it takes roughly-- remember, n is the number of objects in my scene-- p log n time to render my bunny now, because I can kind of traverse the tree of objects in my scene. Of course, notice, I put a question mark here. And the devil's in the details here. In fact, I think computer graphics people often believe that their rendering algorithm takes p log n time. That's often not possible, although kind of an interesting question, which is, the heuristics they use for building these sorts of trees often do, on average, give them log n time.

And so there's something about the data that's making this problem easier than it might seem. So we'll dig into that a little bit in the graphics class. Of course, you're not going to proof as many bounds as you might in a theory course. But we're certainly building on the intuition that we've seen in this class to build on practical data structures. And these data structures appear everywhere in computer graphics. For instance, directed acyclic graphs appear all over the place in computer graphics literature to describe 3D scenes. For example, this classroom is a stark reminder of why we need DAGs and computer graphics, because we have all of these empty seats here, and they're all copies of one another. So would it make sense for me to store however many, like, 100 3D molds of the same chair? Probably not.

So instead, what do I do? I store one instance of a chair, and then some instructions on how to tile it into my entire scene. One way that I can do that is to think of there being a node in a graph which knows how to draw one chair. And now, I can have a bunch of different nodes in my scene for all of the instances of the chair and then store a different transformation for each one. So if you think about the graph structure here, each of those ones is going to point into the same 3D model of the chair for rendering. And that makes a directed acyclic graph structure called a scene graph, which we'll spend quite a bit of time talking about in 837, how to traverse and construct all that good stuff.

And there are lots of different models of computation in that universe, as well. Your graphics card is a very specific kind of parallel processor that's kind of like Lucille Ball on the conveyor belt, hammering at the same object over and over again. But if you ask it to do anything other than the one thing it knows how to do to a bunch of data at a time, then all of your computation grinds to a halt. This is called Single Instruction Multiple Data parallelism, SIMD. Numerical algorithms matter a lot for things like fluid simulation. And approximation algorithms are quite critical, too.

In computer graphics, the complexity is kind of interesting, because of course, your eyeball is sensitive to about 29.97 frames per second worth of material. You can choose that time to do really well-rendering one object, but then you take out of the time rendering something else. There's kind of an interesting conservation law that you have to balance when you solve these kinds of problems, which is an interesting balance, now, between complexity and runtime of your algorithm and perception. What things can you get away with when you draw a scene? And maybe I can do tons of extra computation to get that extra shadow, but it's just not worth it.

I'll quickly sketch out another completely different application of the material that we've covered in 6.006 from my own research. Again, just like Erik-- I guess, in a funny way, both of our groups, I think, are kind of broad in terms of subject material, rather than-- some of our colleagues have really laser focus on one topic or another. Another Research area that I have sort of backed into is the area of political redistricting.

This is relevant in the United States. Recently, I've been reading this great proposal about other countries, which is really interesting, how they do this stuff. In the US, when we vote for people in Congress-- by the way, not necessarily for presidents. This is a common misconception. But certainly for Congress, your state is divided into little regions, each of which elects one member of the House. And there's sort of a subtle problem if you're not used to thinking about it, or one that's staring you in the face and screaming, depending on how often you read the news and politics.

There is an issue called gerrymandering, where your legislature draws the lines for what area on the map elects a member of Congress. And depending on how you draw the lines, you can engineer different results for who's likely to get elected. So for instance, maybe there's some minority. I can cluster them all together into one voting district. Then they will only get the opportunity to elect one person. But maybe, if I divide the space where they live into two, I managed to engineer two districts with a high probability of electing somebody with their political interests in mind.

It turns out that political redistricting, in a broad sense, is a great problem, computationally. Even if you're a totally heartless theorist, there are some really fun problems here. So for example, the state of Iowa-- we all pick on Iowa because it has a unique law, which is that districts have to be built out of counties, which are much larger than the typical census unit, so it computationally is easier. But even in Iowa, which is a giant grid-- with the exception of one shift in the middle, which is fascinating to me-- I know [INAUDIBLE], fun fact.

Literally, people were making the map of Iowa, and they worked from the bottom up and the top down, and it meets in the middle and their grids were shifted, and now we're stuck with that. And it has an interesting effect on the topology of the graph, because it looks like squares, but then there's triangles in the middle. But in any event, even though there's only 99 counties in four districts, there's approximately quintillions of possible ways you can divide that state into four contiguous districts that satisfy the rules as they were-- at least, if you read the code literally in the law.

It seems like computers are useful, but unfortunately, it's a little subtle how. For instance, there's no single "best" districting plan out there. I can't think of a single state with a law that gives you an objective function, similar to whatever cute characters that we've had in 6.006. They often have very clear objectives in life, but unfortunately, redistricting, that's very rarely the case. You have to balance contiguity, population balance, compactness, all of these different things.

Reality check number two is that, even if somebody did give you an objective function, for just about any interesting objective function, it's very obvious that generating the best possible districting plan is NP-hard. And by the way, it doesn't even matter, because the law doesn't say that computers have to draw the best districts. Even if P equals NP really could extract the best possible districting plan using an algorithm, it doesn't mean you have to use it, at least the way the law's written now. Interestingly, this is not true in certain parts of Mexico, where they actually make you compare your districting plan against a computer-generated one, which is philosophically really interesting, although in practice, it doesn't work terribly well.

Our researchers studied analysis of districting plans instead. So instead of running a piece of software that takes in your state, draws your districts, and then you're done-- instead, we ask statistical questions about, I propose a districting plan, what does it look like relative to the space of the possibilities? So that, of course, begs the question, what are the possibilities? So these are the connected graph partitions. Meaning, you have a graph, and you take the vertices and you cluster them together in a way where they're connected to one another. The one thing that we all agree on-- actually, philosophically, it's questionable why-- is that you should be able to start at any point in your district and walk to any other one without leaving. These days, with the internet, it's not clear that that's actually the best criterion. But that's a law that, I think, is never going to get passed in the near future.

Anyway, I think I'm out of time, so I don't think I'll walk you guys through the theory here. Maybe I'll leave it in the slides. There's a sort of very simple proof that can show that, at least the very simplest thing you might think of for analyzing your districting plan, which is to say, you propose a plan, and now, I want your plan to be at least as good, under some axis, as it's a randomly drawn one from the space of all possible connected partitions-- all of the possible ways I could draw the lines. Well, then, it might be useful to have a piece of software that could just randomly draw such a thing. So in other words, to draw something where the probability of any one partition is 1 over the number of partitions.

This seems innocent. In fact, there's a number of papers that claim to do things like this. But it turns out that it's computationally difficult, assuming that you believe that P doesn't equal NP. So I'll maybe leave some suggestive pictures in the slide that we can-- if you guys text me, or if we have a professor-student chat, I'm happy to sketch it out to you then. There's a very nice, easy proof that reduces this to Hamiltonian cycle, and shows you that maybe you shouldn't trust these tools, as much as they're argued about, literally, in the Supreme Court a couple of months ago.

By the way, it was pretty fun. Our expert report was referenced in the defense of the case last summer. And when you read the discussion, you can see the judges trying to talk their way around complexity. And it's an interesting, if somewhat dry, read. In any event, that's just the starting point for our research, which says that, of course, these sampling problems are really hard. The questions is, what can you do? [INAUDIBLE] or not. But the real message here is, of course, that this course is unavoidable. Even in these extremely applied problems showing up in court cases or on your graphics card, you still-- complexity and algorithms and data structures are going to come back to play. So with that, our other two instructors up here for our final farewell-- suitably distance ourselves.

ERIK DEMAINE: So algorithms are everywhere. I hope you enjoyed this class. It's been a lot of fun teaching you and having you as students. Even though you're not here physically in the room, we still feel your presence. And I look forward to seeing you all soon. Thanks for being a part of this fun thing. I want to thank our two-- my two co-instructors for an awesome time this semester. It's been a lot of fun teaching to you guys.

JASON KU: Thanks for spending 006 with us this term.

JUSTIN SOLOMON: Yeah. Thank you. And hopefully we'll see you again soon.

ERIK DEMAINE: Bye.

JASON KU: Bye.