PROFESSOR: So today's class we talked about vertex unfolding, unfolding orthogonal polyhedra, and something else which there weren't any questions about-- got them already-- Cauchey's rigidity theorem. So we will go through those in turn. A bunch of small questions and two cool new things, new updates.

So first question is about vertex unfolding. Why do we fold things in a chain and kind of a line, linear style? Would you get any benefit out of a tree? The answer is chains are easy to avoid intersection. That's the reason we use them on these kinds of unfoldings.

We could dedicate a different slab for every piece and this is great. And incidentally-- I didn't mention this in lecture, but it's obvious once you think about it for five minutes. If you want general cuts-- if you're allowed to cut anywhere on the surface and you want a vertex unfolding-- then anything is possible because you could take any surface and chainify it in the same way we do with the [INAUDIBLE] sections.

You subdivide. First, you triangulate. You subdivide each triangle into three triangles. Then you can hinge it together in a chain. And then you can just lay it out like this. So general vertex unfolding is trivial for this reason.

When you're only allowed to cut along edges, things get much harder. The only technique we have right now is facet path unfolding. But of course, vertex unfolding convex polyhedra is an easier version of edge unfolding context polyhedra, but we have no idea how to do that.

So maybe it's possible. We already know it's not possible with chains. So if it's possible, it's somehow possible with trees, I guess. We know tree style unfoldings can be helpful with general cuttings, but this is not directly relevant to vertex unfolding. But this is the source unfolding and the star unfolding again. So yeah, who knows? But so far, chains are just a useful proof technique.

All right, next question. This is vertex unfolding summarized in the lecture notes on

the right side. So we end up with a nice Eulerian graph. In this case, every vertex happens to have even degree. In general, two of them might have odd degree and we just get a path.

But the observation in this question-- maybe I'll go to the other side so you can read both-- is that we visit the same vertex multiple times, like we attach these two triangles at this vertex. We also attach these two triangles at that vertex. Is that OK? Yes, we define it to be OK.

I think I briefly mentioned it's defined to be OK. Why? I don't know. It's hard to imagine little dots of paper there, but somehow you imagine that they can stay connected. In particular, it's the natural definition of a hinge dissection. You could add hinges at those two places.

But also it's impossible otherwise. This was a fun little puzzle to think about. But if you take-- I actually have a slide for it. I don't have to draw. It's hard to draw an icosahedron. If you take an icosahedron, it has 20 faces. And the dual is a dodecahedron so it has 12 vertices.

So there aren't enough to go around, at least if you wanted a chain unfolding. Well, actually any tree unfolding, if you have 20 triangles, you're going to need 19 vertices to connect them together. And there's only 12 vertices so you can't do it. Is that literally true?

I guess you could imagine, in a general tree unfolding, you could use one point to connect together many triangles. But I'm pretty sure it's not possible if you're allowed to reuse vertices. So that's why we allow it. We get a more interesting result when we allow revisiting vertices.

Could be still an interesting question like, which polyhedron have vertex unfolding if we only get to use the vertex once? But I don't know any results on it. Next question is, any progress on vertex unfolding? And there's one new result.

Before I showed you, I think, one example of a vertex ununfoldable polyhedron-- this guy, box on a box. Initially, we said oh, this is edge ununfoldable, but it's also

vertex ununfoldable because this face here is a donut and the five sides of that cube have to fit in here, and there's not enough area for them. And that's true also if you're vertex unfolding.

No matter how this stays attached to the donut face, it's got a line here, that's not possible. But this is an unsatisfying example because you have a face that is not a disk. And in general, you're not topologically convex. you can't just move the vertices around and make this a convex polyhedron, not strictly convex anyway.

So here is a new example with Zach Abel. He took this class two years ago, or five years ago, I guess. This is a topologically convex polyhedron, so you could move the vertices to make this convex. That's not so obvious. And it is vertex ununfoldable, which we will prove.

This is one of the two results for today. The example is a dark prism here, a triangular prism, and a light triangular prism just at right angles to each other. Then you take the union of that solid, and then you take the boundary. So first, you can check all the faces are disks.

And for topological convexity, you want to check that if you look at any two faces like this light S1 and this dark one, they should only share one edge at most. They should share nothing, a vertex, or an edge. That's topological convexity because that's how things work in convex if the polygons were convex. So like here, they share just that edge.

So it is topologically convex. Let's prove that it is vertex ununfoldable. So in some sense, this is a strengthening of the witch's hats and stuff because this is also edge ununfoldable. But it's even cooler. It's vertex ununfoldable.

Let's just look at the connections between the dark and the light. There are one, two, three, four vertices on this side and four vertices on the backside that connect dark to light. I should make some *Star Wars* reference I suppose. All those vertices are symmetric, so we can just look at any of them.

Somewhere, the light side has to be attached to the dark side. Where could that be?

Let's say vertex A because it's got to be somewhere, and it's symmetric. So vertex A has three angles incident to it. There's alpha-- that's this angle-- which is big. It reflects. There's this beta and there's this gamma.

Beta is 90 degrees here because this is supposed to be a rectangle. Gamma is well, let me state what I want from these angles. There's some parameters in the construction I haven't yet told you. So we want alpha to be bigger than 270. Theta equals 92. And we want alpha plus gamma equal to 360.

So this is possible if you angle these guys correctly, is the short version. I guess the most interesting is this alpha plus gamma equals 360. Is that obvious? Not especially, but I'm claiming it's true in this picture. We'll see it in an unfolding, actually, on the very next slide.

Yeah, let's go there. Still have the same picture. But when unfolded, it's supposed to look like this. So what happens at A? If you want to connect the light side which is alpha that's forced, you can attach by a vertex unfolding to beta or gamma.

Well, beta is impossible because 90 plus greater than 270 is greater than 360. So you would get overlap locally at this vertex if you attach these even if you're vertex unfolding. So you've got to attach the gamma. Alpha plus gamma is supposed to be exactly 360.

And so vertex unfolding doesn't buy you anything. In order to not locally overlap here, they must fit right like that. Then the only other condition we need is that these prisms are long enough so that you get overlap over here. So it's not a purely local overlap, but it's very close by. It's the same two faces overlap. Here, they happen to overlap here and here.

If this were shorter, if this were like this short-- this is the length of the prism here-- then you'd avoid overlap. And that's this picture. So this is an unfolding of one of the two prism halves. This is unfolding of the other. If you attach them, these vertices as is necessary, you get overlap. Unless the prism were short, then you would get this kind of unfolding.

So that's the proof. You can actually make the proof even simpler, kind of nifty. Although it's less obvious that this example exists. You can give the coordinates and you can check it exactly and it has all the properties you want.

Each prism has been bent a little bit. More or less the same example, but now it has the feature that alpha plus beta prime is greater than 360, and alpha plus gamma prime is greater than 360. So you just can't attach anything. Even locally, you get pure overlap.

So easy to check if you're a computer. Easier to check and proof and verify if you're a computer. Less easy visually. So that's what's known.

Open problem-- which I should probably write down-- is if your faces are convex polygons, is there a vertex ununfoldable polyhedron? That, we still don't know. So the witch's hat, all the faces are triangles. There's no edge unfolding, but it has a vertex unfolding.

These guys are nice. They're topologically convex, but the faces are still very nonconvex. And we use that because we want a giant angle, a reflex angle together with something to add up to more than 360. If you're always adding up two convex angles, you don't get that.

But maybe you could combine it with a polyhedron that's not Hamiltonian, so you have to have three things joining it together and then you could get more than 360? I don't know. It could be an interesting problem to work on. But so far, it is open. So that is vertex unfolding.

Next question is about general unfolding of nonconvex polyhedra. Still it's open whether every polyhedron, say without boundary, has a general unfolding. Still we know that all orthogonal polyhedra of genus zero have an unfolding. That remains the state of the art.

But we have a better way to unfold orthogonal polyhedra of genus zero. And this is a new paper. It's not yet online but it will be shortly. The final version just went to the

journal. "Unfolding Orthogonal Polyhedra with Quadratic Refinement."

So two of the same authors-- Mirela Damian and Robin Flatland from the exponential unfolding. You may recall, we saw a solution in lecture that has exponential refinement. The number of cuts in particular can be exponential in n in the worst case.

This requires you take the grid. You refine it, at most, quadratic number of times in each dimension. So the number of cuts is polynomial, I guess at most, and to the seventh if you've got n squared by n squared by n squared by n. I think it's actually much less than that. Probably more like n squared, but I need to be careful.

So I want to cover this solution. It's a simple variation actually on the previous one, but also there's some questions about how the exponential one works. I want to go through that one more time. We have new figures for it from this paper. So it looks a little different, maybe a little bit clearer. That was our girl.

I need my hands. Remember, the general idea was to slice your polyhedron in one direction so you end up with these bands, strips. And then you visit the strips in a treelike fashion.

And in particular, at the leaves, like the closest to you, you've got a band and then you've got a front face. And so that would look something like this little box. So you've got a band that goes around-- which way is it? Band is this way it. Should be an x and z.

And then there's the front face which here is drawn on the back. Those are the five out of six faces of the box that are actually present. This face that's closest to us is actually attached to another band. You could have leaves on this side or that side.

So what do we do? We enter at S and we exit at T. And we are told that the first turn you make from S should be a right turn. And the last turn before T should be a left turn.

And you start in the unfolding, you will start going vertical on S, that's why you

should turn right because we never want to turn left. We never want to go left in the unfolding, which is here. And we also want to end in the vertical state for T so that we can chain these together.

So we started S. We're told to turn right, which we do here. Then we're basically going to zigzag. And if you check along this surface, we turn right. We go here, up there. And then we turn left, and then right. And then this is a left because we're supposed to be doing it from below.

And then a right with that red thing, then a left, right, down here, and a left, and we're done. And the number balances out, so we start vertical, we end vertical. And then when we're done with all these things, we thicken. It's not drawn here, but these strips get thickened.

And you end up attaching these fatter versions, but basically you're following the same path. We don't thicken it at the beginning because we're actually going to do this many times on the same cell. And then we abstract that picture to this iconography which says-- this I didn't talk about much in lecture, which says, you should start by turning right in S, and then you should end by turning left and then visiting T.

So that's one picture. But it depends where you're going. There are actually eight different versions. I showed you this one. But all those combinations have to happen. And the top ones are when you're all the way forward. The bottom ones are where you're all the way back in these slabs, so that's slightly different.

And then there's whether S is on the left or S is on the right, relative to T. And then there's whether you start by turning to the right from S or end by turning to the left. T is always the opposite. But those are the eight possibilities from three different choices.

Are you far forward? Are you far back? Do you turn right from S or do you turn left? And is S to the left or is S to the right? That's why you have eight combinations. They all look basically the same, but those are your base cases.

So this is still all review of the old version. And then the last part of the old method is non-leaves. So in general, you have a band which is here drawn as two separate pieces, but really this is one band. The bands are easy to unfold. They go straight.

But then you have all these children. So you have your parent which is the S and T part. You have to start here and end here. Recursively, you want to visit this whole subtree. So you might have children that are on the front side, children that are on the backside.

And again, we're told we should initially turn right. So if we turn right, we visit the first thing. We tell this leaf or this child, actually-- this is not necessarily a leaf. We tell this child that the next thing we want to do is turn to the left because we just turned right. Actually, we just turned right twice, so we've just gone right, right.

We'll actually be unfolding initially up, so we turn right twice. Now, we better turn left and then we will end up pointing I believe down, because initially this was pointing down. After you do this construction, you'll be pointing down again. And that's why you want to turn left here.

So you unroll this thing. You can verify you are actually just zigzagging. And so you go back and forth through these. Verify this always proceeds to the right. But we're using different iconography here. This is an S on the left that initially turns left. This is an S on the right that initially turns right. That's why we need the different forms of the recursive call.

This is a picture where initially we turn right on S and S is to the right, but there are different versions of this picture just like in the base cases where S is to the left and where you initially have to turn left. But they look the same. You just visit them back and forth like this.

At some point, you finish the front children. Then you go up. And here, it's a little bit shifted but these would normally be aligned. Then you have to visit the top children. You do that by visiting and going back and forth.

Then, of course, when you're done here, you have to unwind everything you did.

Then we go over here. We visit that guy, back and forth. Eventually, we will end up on the inside here, and having doubled everything, we come back out to T.

And all the time, we're unfolding to the right. Now, what can we say about this? Well, what sucks is all of these children get visited twice. And that's why you're doubling at every recursive level.

And so if your recursion tree looks like a long binary tree, then you get exponential. Not quite. We'll come back to that. Here's an example of it being exponential with this kind of tree. It's a little hard to draw too many levels deep.

So we have one level deep and then another level deep. But you'll notice that it's not uniform. This leaf gets visited exponentially many times, but this one still only got visited once. Of course, it's shallower.

But even these two leaves are at the same level and they did not get visited the same number of times. So there's some hope of improving things as you might imagine. So this is where we go.

There's two observations here, two ideas. Maybe I'll tell you where we started. I was I think-- I can never remember years. I was teaching this class either two or five years ago and I had idea number one. And we worked on it in the open problems session, but it was not enough. Idea number one is just part of the story.

And it's an idea that comes from the other graduate class I teach which is heavy-light decomposition and advanced data structures. So this is a very common technique in advanced data structures. It goes back to Sleator and Tarjan, if you've ever heard of splay trees or link-cut trees.

It's that era which was the '80s, I believe, '83. It's a very useful tool. Whenever you have general tree structure and you want to make it balanced, you should think heavy-light decomposition. There are actually a few ways to balance a tree, but heavy-light decomposition is usually the most powerful.

So essentially what it does is say if I could just treat this thing, this path as a single

node, and then treat these guys as separate nodes, then I'd kind of be balanced. It would be like one super node and then a bunch of children hanging off of it. One, two, three, four.

So that's the picture I'd like to make. I'm going to contract paths, only paths of my tree. And it turns out there's always a way to take paths, contract them into single nodes, so that the resulting thing has height at most log n, no matter what tree you start with.

And heavy-light decomposition is a way to do this. It's very simple to define. For every node, you define the idea of a heavy child. This is a child with more than half the weight. The weight of a subtree is the number of descendant nodes, so the number of nodes in that subtree.

So if we look at this picture, you've got this node. It has one, two, three, four, five, six, seven, eight, nine nodes below it. This subtree only has one, so this subtree has seven. Seven is bigger than half of nine. Most of the weight is to the left.

So we call this edge heavy. Similarly, this edge is heavy because this is five. Five is more than half of seven. This one's only three nodes below it. That is still more than half because half of this is 2.5.

So this edge is heavy. This one is not heavy. So actually, in the real picture, it would look like this. This is a heavy path and this guy's all by itself, so there's one more node. But it has the same effect of balancing the tree.

So what's the point? In general, I have some tree. In general, I might have many children. No idea how many children are out there. So every node can have many children and these correspond to the various bands that are attached to my parent band. So that's heavy child.

In general though, every node has at most one heavy child because you can't have more than two halves. If I have more than half of the weight in one child, I can't have more than half of the weight in another child. So every node has at most one heavy child.

These edges connecting nodes and their heavy children we call heavy edges. And whenever we have a node, there's at most one heavy edge going down from it, so heavy edges form heavy paths. That's the point. So this red thing is called a heavy path. Cool.

So what do we know about heavy paths? The other edges, I should say, these guys, we call light edges. This is the heavy-light decomposition, so every edge is either heavy or light. It could be a node only has light children because it's well-distributed.

In the worst case, it has one heavy child, one heavy edge coming out of it. So we get heavy paths. I guess I'll say heavy edges form paths. And if we look at what I normally call the light depth of a node, this is if I look at a node in this tree and I measure how many light edges do I have to traverse in order to get from the root of the tree to that node, here it's only one light edge.

The number of light edges follow on any path is at most log n, log base 2 of n, because every time I follow a light edge, I know that I started with say end nodes. I follow a light edge. I know that I have at most half n nodes because my weight goes down by half.

If I start with n, I can only reduce by a factor of 2 log n times before I run out of nodes. Then I'm in a leaf when I get down to the weight of one. So light depth of any node is at most log n because weight halves every step, every light step.

So this is why, if you contract the heavy paths to single nodes, the depth of the remaining tree is only log n. So what does this tell us? If I could somehow deal with paths efficiently and not feel the recursive pain that we're getting here from doubling everything along a path, then I can chain them together in the usual bad recursive way. I can afford to double on a balance tree.

So if I could somehow treat these paths as single things, we'd be all set. That was idea number one and that sat for years. I knew this should be possible. But what we

were missing was idea number two.

It's funny. If you stare at this picture long enough, it's actually kind of obvious. It requires more work after the idea. So here, the same picture as before. I spread it out a little bit.

So we end up visiting this guy last. And then we end up recursively visiting this one a second time, then this one a second time, then this one a second time. We go down here. We go around here.

Visit this one a second time. Unwind. Visit this one a second time, this one a second time, this one a second time, and we're up. Did I visit them all a second time? No.

There's one that I didn't visit a second time, which is the last child. I only visited that one once. And that's all we need is one child because I only need to do a path efficiently. So I'm recursively visiting most of the children here.

Those are going to be my light children. The one heavy child, I want to aim to visit that one last because that one's only visited once so it doesn't get recursively doubled at this level. Of course, the higher up the level, it will get doubled. But there's only log n levels we're going to get doubled, so we'll only add a factor of n.

Idea number two is that the last child gets visited only one at this recursive level. And if you've taken an algorithms class, you know recursively visiting things once versus twice makes a big difference. Here it's a matter of doubling or just staying the same.

If we can rearrange to visit these children first, then do this one last, then visit this child first, then do this one last, then visit this child first, then do this one last, and then do all the children in whatever order, then we'll guarantee that this path doesn't get recursively refined at this level so it's really being treated as a single node basically. And that lets you tie together these unfoldings in an efficient way. The only issue is I need to be able-- this construction isn't enough because this construction basically forced this one to be the last child to get visited.

I need to be able to control which child is last visited because the heavy property is a property of the tree. One of these guys, maybe this one is heavy. Maybe that one's heavy. We've got to be able to end on that.

So there are two cases. First case is that we always start from the bottom here. And first case is that on the top side is where the heavy child is. So this is the easy case because the bottom, we don't care. We just visit the children in whatever order's natural.

Then we come up. And basically, we have the freedom to spin around here. We have to turn left, but we could either choose to immediately go up here or keep going and go up here, or keep going and go up here. That would visit that one too early. Or go here.

Turns out this is the right choice. But we had freedom to just spin around and choose when to go up. Just try them all. One of them will end up being last. Basically, you want this guy to be in the middle between where you start and the one just to the left here because you're kind of rainbowing back and forth, zeroing in on the last guy.

And just by pigeonholing, I suppose, one of these choices will work because there's k choices you could make on when to turn up that each result in a different guy being last. And so one of them corresponds to the correct one. That's easy to figure out which one it is. So if it's on the top side, it's really easy. You just pick it.

This would be great. This would actually give us linear refinement if we only had this case. The other case is when the guy we want to make last, the heavy child, is on the bottom side. In this case, we need to do more work.

So this is basically the old construction. Here we don't really have freedom from S. We can't just bypass this guy because it will actually be hard to get him later. So we've got to visit him and back and forth. But the one thing we could do is just omit a child, just forget about it for a while. Don't visit it. Visit everything else in whatever order feels natural to you.

And then we're going to splice this guy in. So we'll see where does the path go. At some point, it will go above this proposed last guy. And what we're going to do, which is on the next slide, is on the return trip-- so remember we have this picture. We come out here. Then we recursively double everything.

So at some point, we are coming here, doubling. And at that point, we turn in. But if S and T are aligned like this, we are pointing the wrong way. We want it to get here, but it doesn't even look like we're in the same side. But we are. We just have to recursively double everything again.

In the end, when you recursively double everything again, you will be here. And we haven't finished doubling here. But in the end, you will come out here and exit at T.

So overall, these guys which haven't been visited yet, they'll end up being doubled from this procedure. These guys will end up being quadrupled. So we're doing four recursive calls for most of the children. Some of them might get lucky and only double. But all we care about is if this guy only gets visited once. That's the key.

Because this will be recursively here. It's hard to unwind. But then you'll end up on the inside of this channel. Then you'll end up visiting this guy. Then you'll end up visiting this guy. And then you get out at T.

So you won't come back to here. That's the key. That's all we need. You can write down a recurrence at this point. If you look at the level of refinement for an n vertex polyhedron or an n slab polyhedron, it would be-- n, we'll say is the number of nodes in the tree of slabs, this picture. In general, that is proportional to the number of vertices in the polyhedron.

In the worst case, it's going to be the max of two things-- R n minus 1, and 4 times R of n over 2. Well, I should say n sub i, max over i. So what I mean to say here-- this is also technically an sub i. This is an upper bound certainly.

So if we look at the recursion, how much refinement we need, the worst refinement over the whole tree is the max of all of its subtrees. So there's a [INAUDIBLE] here, a [INAUDIBLE] here, and then also of course, what we need to do within the node.

And on the one hand, there's the heavy situation, the heavy edge.

That only decreases the size of the tree by 1. But it only gets recursively visited one time, 1 times R of n minus 1. There's the other light children. These are all light children. n sub i represents the number of nodes in that child. We know that this is at most n over 2 because we know that at most half of the weight lives in the light child.

Now these get recursively visited at most four times, up to four times. Worst case, it will be four times. In this recurrence, this first term doesn't matter. R of n is at most R of n minus 1. Yeah, big deal. That's just saying R of n increases with n.

So really, this is going to be-- and then the max disappears. All we need is this is at most 4 times R of n over 2. And this is a recurrence. If you read CLRS, this follows the master theorem and you know immediately this is order n squared. It'd be theta n squared, but this is an upper bound.

In the worst case, it will actually be theta n squared. So the amount of refinement is at most n squared. If this is not obvious to you, an easy way to imagine it is you start with something of size n, you're visiting four times something of size n over 2. This is called a recurrence tree, recursion tree.

Each of these visits four times something of size n over 4 and so on. Obviously, log n levels here. And at the bottom, you're going to have a bunch of ones. That's when you stop recursing. And if you add up everything level by level, how much work am I doing here and how much work am I doing here? 2n.

How much work am I doing here? It gets harder to add, but there are 4 squared, 16 things, each n over 4. So it's 4n. In general, it goes up by a factor of 2 every time which means you're dominated by the last layer. And if you count, it's going to be n squared because this is 2 to the 0, 2 to the 1, 2 to the 2.

After you get to log n, it's 2 to the log n times n, so that is n squared. And this is a geometric series if you want to add all these numbers up. And so it's 2 times n

squared. So that's the proof of this bound and that's how you get quadratic refinement.

So either idea is kind of easy, but the two together, pretty powerful. Questions about unfolding? I think this algorithm should be pretty clear. One of the questions was, are they practical? I think the answer is no.

It has the same issue of strip folding from origami. But I think they would make great animations. This is still a cool project to see virtually. But to do it physically is probably crazy. No one has tried as far as I know.

Another idea. This is an open problem. So orthogonal was great. It has all these nice properties. Can you generalize it, not to arbitrary polyhedra, but to some other kind of orthogonal, such as this is a hexagonal lattice in 3D? I mean, it's just extruded in one dimension, but then you have hex grid in another dimension. You could try to do that.

And my guess is if you set up the bands to be in hex dimensions, it just works, but I haven't tried. There's potentially some low-hanging fruit here. There should be other 3D structures that might work.

A typical one in computational geometry is called c-oriented polyhedra where here we have three oriented polyhedra. There's three different directions that the faces can be perpendicular to. Take 10 of them, can you apply the same technique? That gets dicier.

But something like this I feel like may be possible. You should try it out. But as far as I know, this is all open. Last question is Cauchy's rigidity theorem seems intuitively obvious. Or as I wrote it in the notes, is Cauchey's rigidity theorem obvious? And my answer is no, not obvious to me.

I have two reasons why it's not obvious. One is it's not true in two dimensions. That's kind of a weak statement. But if I take a convex polyhedron in 2D-- so these are rigid faces. This is the equivalent of a triangle. This is not rigid, of course. So that's one statement.

And then the other thing is convexity is necessary. There's this thing, Steffen's polyhedron. Actually, originally, there was a Connelly polyhedron. We've seen some results by Connelly. He's a rigidity expert.

And then this was simplified by this guy Klaus Steffen in the '70s. This is a drawing of it flexing in 3D. Nonconvex polyhedra can be flexible. They happen to preserve their volume as they flex.

It's hard to say why a theorem is not obvious. To me, it's not obvious. Of course, if you play with convex polyhedra, they are rigid, so maybe it's obvious from that. But you've got to prove experimental evidence mathematically. Other questions?

Then I have some fun videos to show you. The first one is the making of this origami piece which I've probably shown a picture of before. It's by Brian Chan. It's one square paper, no cuts. *Mens et Manus* with a little crane instead of-- what's it normally? Who knows? And then he also made this nice box. And then the glass is etched by Peter Houk who runs the glass lab at MIT.