PROFESSOR: All right, so lecture 11 was about generic rigidity of linkages. So we've got bars and vertices connected together in a graph. And generically, the graph is all that matters. And we characterize in two ways, Henneberg construction and Laman when graphs are generically rigid.

And in particular, there was this pebble algorithm that was supposed to tell you in polynomial time whether you're a graph was Laman or not. And everyone pretty much asked about this, about the details, including myself when I watched it like, huh, that seems a little sketchy. So I thought I'd spend most of today going through how that algorithm works exactly and also why it's correct, which is definitely not obviously. It vaguely feels correct, but we'll see why it actually is.

Remember Laman's Theorem, Laman characterization is that your graph is generically rigid if and only if every subgraph-- let's say every k vertices induces at most to 2k minus 3 edges. That's part one. And you also need that the number of edges overall in the graph is 2n minus 3 where n is the total number vertices.

So this was just the degree of freedom count. You have 2n degrees of freedom for every vertex minus 3 for the rigid motions, two translations, one rotation. So that's the number of edges you should have total. Sorry. This is for minimally generically rigid. And then to make it balance, you want to guarantee that every k vertices doesn't have too many edges. Because if it has too many edges, there'd be too few somewhere else. This part would be overbrace, some other part would be underbrace, and you'd be flexible. That's the intuition.

And so the tricky part comes down to how do you check whether every k vertices induces 2k minus 3 edges because they're exponentially many choices. There's 2 to the n different choices of subsets or vertices. For each of them, you'd have to check. That's no good. So the algorithm wants to check this. And I'm going to call this property the 2k minus 3 property.

And we're going to warm up with a simpler problem, which is to check to the 2k property without the minus 3. So 2k property is just every k vertices induce at most 2k edges. So this'll be a little easier to think about, more or less the same, a lot easier to argue about. And then we'll see how to modify the algorithm to check the 2k minus 3 property.

Was the property clear and in particular inducing? Inducing means that you have some set of vertices, and there's the rest of the vertices, n minus k of them. You only look at edges where both endpoints are in the set, both of them are among the k vertices. So you ignore vertices that have one end in k and the other end outside. And you ignore, obviously, edges that have both ends outside. Just count these guys.

So how do we do this? Give you the algorithm. Actually, let me first give you the idea of the algorithm before we go to the actual algorithm. And this is review, but pretty soon we'll get to stuff that wasn't covered in lecture. So the idea is every vertex has two pebbles inside of it. I'll draw as two red dots.

And they can cover incident edges. So they can't move very far. They have to stay basically on the inside or on the boundary of the vertex. That's how I'm going to draw it. Each pebble can cover one incident edge. So for example, there's a little graph. I could take-- let's maybe I leave one of the pebbles in the middle. I can move one of the pebbles to cover one of the edges.

So I'll put it this intersection to denote it's covering this edge, but it still belongs to this vertex. Pebbles cannot change vertices. They have to stay local. These two pebbles could just stay where they are. Here, we could put one pebble here, another pebble there, one pebble here, one pebble here. This is what I call a pebble cover cause every edge is covered by some pebble.

That's the goal in a pebble cover is to cover every edge. You need to cover it once. And the pebbles can either cover an edge or they can be free. Free if they're just hanging out in the inside here not being used for anything. So our goal is to find pebble covers. But the first claim, why do I care about pebble covers, I claimed that having a pebble cover is equivalent to the 2k property. So that's what I want to show.

So let's start with that. This is why everything is working. A graph has a 2k property if and only if it has a pebble cover. So that's why we care about these pebble coverings. And this is actually pretty easy to prove and gives a lot of intuition for why everything works here. So let's start with the left direction.

So suppose I have a pebble cover. Suppose somehow I've achieved this property, covering all the edges, using only two pebbles per vertex. I claim that we then satisfy the 2k property. 2k property says you take any key vertices, you induce at most two key edges. So let's consider k vertices. Someone chooses k vertices. doesn't matter which. It should hold for all of them, for all choices.

So we've got a k vertices here. n minus k vertices remaining. We claim that the number of edges induced by those k vertices is at most 2k. So look at one of these edges that's wholly inside, that's induced by these k vertices. I claim, well, that edge must be covered by a pebble. Which pebble? Well, either this one or one on the other side. But the point is it is covered by a pebble among these vertices.

My concern is, well, maybe you worry about an edge like this. Or, maybe you worry about a pebble here that's covering this edge, but that edge is not one of the induced edges. So these vertices are relevant. It's just the pebbles inside, just among these 2k pebbles. So in here, there are 2k pebbles.

They must fully cover the induced subgraph. They must cover all the edges in here. That means they're at most 2k edges because there's one pebble per edge at least. And so there's 2k pebbles in here. This means they're at must 2k induced edges. Clear?

That's the obvious direction. This doesn't really exploit very much about pebbles. The other direction is a little more interesting. Why should these pebble covers exist? Because they seem pretty constrained. Pebbles can't move very far. It's not really clearly the same, but it is. So if we have the 2k property--

So now we know that every subgraph of k vertices induces at most 2k edges. Now, we want to prove that a pebble cover exists. I'm going to wait. It will become obvious momentarily. This is fun. Sometimes, it's easier to prove a theorem than it is to come up with an algorithm. But in this case, it's easier to come up with an algorithm than it is to prove a theorem.

And you can use the algorithm itself to prove the theorem, so I'm going to prove this in a moment. Stay tuned. Meanwhile, we have an algorithm. This algorithm is an example of what we call an incremental style algorithm where you imagine starting with the empty graph and then you put in all the edges that belong one at a time. So add edges one at a time starting from nothing.

One detail is we're going to imagine as edges get added, we want to direct them. We're going to direct edges from the pebble that covers them to the other side. So if we have an edge and we have a pebble over here, than I'm going to imagine the edge as being directed from the pebble to the other side.

That's just for convenience for expressing the algorithm in a moment. So now this is for the 2k property. We'll get to an algorithm for 2k minus 3 a little bit. So now we have a 4 loop over the edge is we're going to add. And when I immediately add an edge, it will not be covered, so it's not directed, just some edge v-w.

I would like to put a pebble here, but it could be both pebbles at w are occupied for some other edges. It could be the pebbles for v are also occupied for some other edges. That would be annoying. I want to somehow reassign a pebble to cover this edge, so what I'm going to do inside this 4 loop is search for available pebbles.

And in this case, what turns out to be right is a directed path in this directed graph from either v or w to a free pebble. Either I find one or I don't. If I find one, let's say from v, there's some edge and then some other edge and some other edge. And eventually, I find a free pebble. Remember a free pebble is one that's floating on the inside, not being used for anything.

So what does my picture look like? Remember edges are directed from the pebbles. That means there was a pebble at v being used for that edge. There was a pebble at this vertex being used for that edge. There was a pebble at this vertex being used for that edge. In this case, what do I do?

AUDIENCE: Reverse.

PROFESSOR: Reverse them, yeah. Just reverse all the edges. So I have this edge, v-w, here. I want to put a pebble here, so I'm going to reassign this guy to be over there, this guy to be over there, this guy to be over there, and so on. So I end up with-- that's in the way. Put w over here. The new picture is I have v and w. And I'm going to have left 4 directed edges.

So now this pebble got assigned to be here. I can't draw left apparently. The other left. This pebble assigned there. This pebble, the free pebble, gets used up. And now we've got a free pebble over here. And now we can assign it to be on the edge v-w. So if I can find such a path from v or from w, I'm happy because then I can reassign.

If I find such a path, I just shift the pebbles like that, and I'm done. But what if I can't? This is the critical issue, and it is also core to this part of the proof that we haven't discover-- haven't yet done. Why would we hope for pebble covers to exist in the first place? We don't know. I claim if this algorithm fails to find a path in this situation, then you can immediately say you do not satisfy the 2k property.

I'd like to say there's no pebble cover, but that's a little harder. I'm going to say you do not satisfy the 2k property. And we actually know already, from this part, pebble cover implies 2k property. The contrapositive, which we've already proved, is if you do not satisfy the 2k property, you do not have a pebble cover. So that will actually guarantee correctness of the algorithm. If I can prove this part, that it violates 2k property, then I'll know there was no pebble cover, and I'm done.

So why is this true. I claim if you look at all the nodes reachable by directed paths from v and w, then that is a set of nodes that induce more than 2k edges. So let's say their k nodes you can reach. I claim there will be more than 2k edges among them. Why? It's fun to have proof in the middle of an algorithm.

So here we've got to v and w, and these are the k nodes that are reachable from v and w, which means all of the other edges are pointing into this set. And then, over here, we have everybody else. I mean it might be everyone. But potentially, there's some vertices over here. All the edges are directed from right to left, otherwise we could reach more stuff.

So what's the situation? v-w is uncovered. There's no pebbles that cover it. Every other edge is covered because there's only one uncovered edge at any moment. These edges are all covered from this end. That's what these directions mean, which means all the pebbles in here are used to cover induced edges.

So all the pebbles in here are covering induced edges, and yet, there's one induced edge that is not covered. Now they're exactly 2k pebbles. They're all being used. I'm assuming here you never have two pebbles covering the same edge, so that means-- so there's 2k pebbles. They're all used to cover induced edges, which implies you have exactly 2k plus 1 edges. Plus 1 is v-w.

That means that means the graph that we've created so far violates the 2k property. Now, we haven't added all the edges yet, but if we add more edge, it's going to get even worse potentially. If we have 2k plus 1 edges now, in the actual graph, we'll have at least 2k plus 1 edges. So if this search fails, you know that you violate the 2k property, which means you know that there's no pebble cover.

So that proves correctness of the algorithm. Now we can come back to this claim. This follows, by correctness of the algorithm, why. We want to prove if we have the 2k property, then you have a pebble copper. Well, if I have the 2k property, this algorithm will work. This case can never happen because it can only happen when you violate the 2k property.

So if you have the 2k property, this algorithm, you can keep running it. You will always find a path. You'll be able to shift the pebbles and always cover things, and so you get a pebble cover from the algorithm. The end. That's how you argue that if you have the 2k property, you get a pebble cover. And it's nifty.

Without this algorithm, this would be quite tricky to specify. But the algorithm tells you where the violation would be, tells you where you would violate the 2k property. And that lets you prove the claim ultimately. Any questions about that. This, I think, is the heart of pebble algorithms and why they work. It's the cleanest part. It gets a little bit messier to do the 2k minus 3 thing, so we'll move to that next.

Maybe before we go there, we can talk about the running time of the algorithm for those so inclined. So the heart of the algorithm is this step, search for directed path for v or w to a free pebble. This is essentially some of the nodes have free pebbles. Those are like good nodes.

But basically, we've got a directed graph. We want to find all the nodes that are reachable from v, see if there's any good nodes that have free pebbles, search from all the nodes from w. This you could do with a depth first search or a breadth first search or your favorite search algorithm. Takes linear time. Order the number of vertices plus the number of edges in the graph.

We do this search v times. Why v times? So we add edges one at a time. We're hoping that we have the 2k property. The 2k property implies that, in particular, there at most 2v edges. So in general, we do this e times. We keep adding edges. We would add e edges to the graph hopefully.

In general, e can be much larger than v in a graph. e could v squared, the worst case. But this algorithm will fail by the time you add 2k plus 1 edges because then you definitely don't satisfy the 2k property over the entire graph, so particularly you won't satisfy it on some subset of vertices. So you will only have to run this loop v times, not e times-- or 2v times.

So this is a little bit smaller, and so this product is v squared plus ve. Normally, we write ve, but this is slightly more precise, which is the same as Bellman Ford algorithm if you've taken algorithms. I'll just mention that there is an algorithm slightly fancier that just requires this term. This is possible. You don't need this ve part. but it's beside the point to our goal, which is just to understand when things are rigid, when they're not in an efficient amount of time.

So this is polynomial in any case. This also the best known algorithm, v squared. Open problem is whether you could do better. Questions? Yeah?

AUDIENCE: Is there more than 2v edges instead of, [INAUDIBLE] that [INAUDIBLE] problem.

PROFESSOR: Yes, there are more than to v edges. Oh, right! So there's done. Great. You can just check in the beginning, are there more than to v edges. If so, you do not satisfy the 2k property, and so you can stop. So that takes constant time to check those cases, otherwise e is small. And then you only take the square time. Thanks. That's key.

So that was the 2k property. Let's do 2k minus 3. For the 2k minus 3 algorithm, it's going to be pretty much the same as this algorithm. So in particular, every vertex is to attach pebbles. Each pebble can cover when it's in an edge, otherwise it's free. Our goal is to cover every edge.

But it's slightly differently. We have to deal with this minus 3 business. And for that we need another claim, which is that a graph has a 2k minus 3 property if and only if g plus three copies of an edge e has the 2k property for every edge, e and g. Should probably say g has.

So I want to know whether my-- ultimately, I care whether my graph has a 2k minus 3 property. This is most of the Laman condition. It's the hard part to check. And 2k property is what I know how to check. So I claim all you need to do to check it-- this would actually be an algorithm. For every edge, turn that single edge into four copies of the edge, add three more, check whether it has the 2k property, remove those extra three copies, take the next edge, add three copies, check the 2k property.

So with another factor of e, which has to be order v, if you're willing to spend v cubed time, you can just run this algorithm for each edge, quadruple it, see whether it still has the 2k property. So that's the connection between these 2 things. Let me prove this claim. And then I'll tell you a better algorithm that only takes v squared time.

Which one's easier? In both cases, I want to take k vertices. And then I'll do in this direction first. So how do we prove this? First, this is actually really easy. Suppose g has a 2k minus 3 property. So if I have k vertices here, I know that the number of edges in here is at most 2k minus 3.

So here's k vertices. Looking at the number of induced edges, it's going to be at most 2k minus 3. What I'm claiming is then if I quadruple an edge, if I add three new edges, I have at most 2k edges afterwards. I think that's pretty clear. 2k minus 3 would be 4 at most. If I add three copies of any edge, could be in this set or outside the set would be even better, the number of edges inside the set will be at most 2k.

So this is in g. This implies there's at most 2k and g plus thrice e. This is non-standard notation by the way. It just means add three copies of that edge. So we would normally draw that like this, four copies of the edge. So that was half of the claim, pretty trivial. The other half is almost as trivial.

So on the other direction, we assume that g plus 3e has this property for every edge. And now we need to prove that g has a 2k minus 3 property for every set of k vertices. So we're looking at that set of k vertices. We want to argue that in g, there's at most 2k minus 3 edges. What we know is that in here, in g plus 3 e for any e-- so it's a little awkward-- but let me start by drawing g.

We want to claim that the number of edges in here is at most 2k minus 3, so I'll put a question mark. So there are 2 possibilities here. It could be either this graph is completely empty. If there are no edges in here, then surely there are at most 2k minus 3 of them, assuming k is bigger than 1, which I haven't mentioned.

But throughout here, I need the k is bigger than 1. You can never have negative 1 edges on subgraphs, so the 2k in the 2k minus 3 property only hold for at least 2 vertices. K is at least 2. So if there are no edges, find I'm done. If there are edges in here, then I can do this trick. So there's some edge. Pick your favorite edge. Call it e in this subset. Produce out of it g plus 3e, which just looks like the same thing, but now it's got four copies of that edge.

This we know has at most 2k edges in here. The original graph is exactly the same except it lacks those three edges, therefore, there's at most 2k minus 3 of them. So it's really the same proof in both directions. Here, you have to use your freedom in choosing e to choose an edge that is induced in this subgraph. You can't choose some other edge out here. Question?

AUDIENCE: So this seems kind of cheap because we're using the graph as a linkage in the end, so adding these extra edges, they all collapse into one.

PROFESSOR: Ah, interesting. Right. So why this feels like cheating because, from a linkage perspective, these 2 guys have exactly the same constraints.

AUDIENCE: Yeah.

PROFESSOR: I agree. It's weird. At this point, this is a device. You could think of it as adding edges. Another way to think of it is this edge has to be covered by four pebbles. That's probably a little more intuitive. So having to cover an edge by four pebbles, essentially, is pinning the edge. Because on the one hand, you have a single pebble just to say this is an edge.

And then you have three more pebbles to eat up the translations and rotations. That's intuitively what's going on. I agree this looks weird. And at this point, we're just playing the graph theory game and ignoring the linkages. That's the short answer why this is OK. But ultimately, intuitively, what's going on is that these three edges are just representing the translations and rotations being eaten.

And they have to be somehow eatable universally at every edge. That's what this is saying. I don't have a great intuition for that, but it is-- pretty sure you do need to say for every edge, not just some edge.

So at this point, you have a polynomial time algorithm, so I'm done. But let me briefly mention how you could make it faster by modifying this algorithm to do the 2k minus 3 situation. Dare I-- I won't literally modify the algorithm. I'll write a new version. But I'm going to use that as a subroutine basically.

And so algorithm 2k minus 3 version. So it's going to be the same style. We're going to add edges one at a time. We'll direct the edges in the same way for each added edge v-w. I'll just write for each added edge v-w. Before I just tried to add the edge and cover it with pebble. Now, I want to cover it with four pebbles. I want to basically add the edge four times.

I'm going to write as above to mean this step. So every time I add the edge, I search for a directed path from v or w for free pebble. When I find it, I do the pebble shift. I do that four times. Then basically, I have four pebbles on the edge, or you could think of them as being four copies of the edge, each has one pebble.

Either I succeed and I get all four, or I don't get all four. On success, this is just a temporary measure. At that point, I'm going to delete three of the copies. Put it back to a regular single edge. This frees three pebbles for future use. So I don't want every edge to be quadrupled. I don't want them all to be quadrupled at once. I want to try quadrupling one edge, then try quadrupling the next edge, try quadrupling the next edge.

But when I'm quadrupling an edge, all the other edges exist as single edges. So I delete the three copies. And then this four loop continues and tries the next edge. On failure-- here I'm going to say something more interesting-- Normally, on failure I'd say, well, you don't satisfy the 2k minus 3 property. Game over.

But you can be a little more sophisticated. Just say delete all the copies of e. I guess there's four of them. Whatever pebbles you can acquire, you give them up. Delete the edge and call the edge redundant. Basically, in this case, you could argue that edge was superfluous for rigidity purposes. You didn't need it.

So the algorithm doesn't just fail. It says this was a useless edge. And it'll tell you which edges are useful, which ones are non-redundant from a rigidity standpoint. And then when you're done, you will be rigid if and only if the number of non-redundant edges that remain equals 2n minus 3.

So that's the last part of the Laman check. So this is giving you a little bit more information than just do I satisfy the 2k minus 3 property. To get that, you just check whether you ever failed. But in this case, you can actually see, ah, these are the useless edges. They're those overbracing edges. Everything else is useful.

And if I end up having enough useful edges to n minus 3 of them, then I know I'm rigid. Otherwise, I'm going to be flexible. So this gives you all the information you want. With a little bit more effort, you could even figure out if I'm flexible, which parts are rigid and which parts can move relative to each other.

This, of course, generically rigid. Finally, let me show you this thing in action. So look over there. This is an implementation by Audrey Lee-Saint John. And so here's a simple graph. You can probably see this top part is flexible. It's quadrilateral. Bottom part is overbraced. This is in 2 dimensions, right? So just triangles would be enough, but I've added an extra edge here.

And this is going to check the Laman condition one step at a time. So we start with 2 pebbles everywhere. We start by adding that top edge. Right now, there are four pebbles on that edge, so it's great. I can just add the edge. Let's say I added it four times. There are four pebbles there. But then in the end, I just need one. So let's skip that step. And now I've got the one pebble from b, and we're happy. So that is its own rigid component.

Next, let's add this edge ef. So this also has four pebbles on it, so we're done. Next, we have cd. That also has-- you could change the order if you think this one's a little trivial. Now is when the action happens. Let's add this edge bc. So currently, it is only covered by three pebbles. But we need a fourth. So we're going to take one of these free pebbles on a and move it along b.

We found that by starting from b and searching out and finding a free pebble. So we just flip that edge. Now we've got a free pebble on b. Now we've got four pebbles on bc, and we can add the edge. But we only actually take one of them. So I'm skipping the part where we add the four copies and immediately remove three of them because that's silly.

Cool. So that's its own rigid component it turns out. Each of these things is independently-- the colors represent rigid components. And right now nothing is tied together. Everything's flexible. Flexible at b. Flexible at c. Let's add this diagonal df. So right now, it only has one pebble, which means we need four pebbles.

So we're actually going to have to get one at f, and we're going to have to get one at d because there can only be two at each. So first, we'll do I guess a search from d, and we immediately find a pebble. We move it down there. Great. Then, we search from f. We find a pebble there. We move it. Boom! We've got our edge, added all four pebbles.

Eventually we're going to run out of pebbles, but right now, we are very flexible. Let's add this other diagonal. This should also be possible. Right now, there's one pebble at e, one pebble at c. We've got to find two more pebbles. So first, we'll follow this edge. Find a free pebble at f. That was easy. Next, we'll look at from c. There's only one way we can go from c. We can't follow the back edge. Got to go forward. Got a pebble at d. Flip it around. Now, we've got four pebbles on c and e, so we can add the edge.

Now, let's add ef. Right now, we've only got one pebble, yet we need four. So let's start by searching from a. Oh, we found a pebble immediately. In general, we would have to follow many edges. So there's a from f. There are two ways we could go. Either one of them would have given us a pebble. Let's flip the yellow edge. Boom! We've got f, one pebble, but we need another one.

Now, we cannot get another pebble from e because we just flipped the edge, but we can get a pebble from d, so we flip that edge. And now we can add the edge af to more edges. It's a little tedious. So next is cf. We're running out of pebbles here. There's only five pebbles total in the system. We need four of them right here because we're getting almost rigid.

So let's see. We've got cf. Let's start from searching from f. There's only one way we can go. We find a pebble. We flip it around. Now we've got c. There's only one way to go from c. We find the pebble at e. And we flip it around. Now we've got the two pebbles. Cool.

So we've got the three pebbles here, which are representing our translations and rotations. The only other pebble left is this one, which is basically representing the degree of freedom in this quadrilateral. So when we go to add the last edge, which is-- so at this point, this is a ridge it component because of the triangles. And we go to add the last edge, we will fail to find enough pebbles.

Let's see. We can find three of them. We can grab the one from e. Or let's see so-- it's not going to show us. But from e, we can grab f, the pebble at f. From d, we can grab the pebble at c. From d, we can grab the other pebble at f. We've got three pebbles, but there's no way to get to this pebble because both of these edges are directed down. So you can never get from the below part to the above part.

So it's not obvious that this is working. That's what the proof is for. But at least you see it in action. I'll just run it very quickly for another graph. Let's say a big one. This one is minimally, generically rigid. So it's fun to watch for a while. You get pebbles moving around. In this case, you have some longer paths you have to follow to get enough pebbles. It depends a lot on the insertion order that you use.

So in each case, you need four pebbles, two on each side. And if so, you can add your edge. Otherwise, you can't. What I haven't described is this color coding which is figuring out when you have rigid components. Roughly speaking, to find what your rigid component is-- if you're an edge, you want to know what rigid component am I in-- you basically just search for all the vertices that are reachable from your edge. Those are all in your rigid component.

That's what we saw. So this graph happens to be minimally rigid, and it detects that. But then it can be a little bit more as in your rigid component. Some of the incoming edges-- let's see. Draw a picture. So here's everything you can reach from v-w. And say you're following an incoming edge here from some other vertex u.

And then suppose these are all the things you can reach from you. If there are zero pebbles out here, then this whole thing is a rigid component. And you just keep doing that. You check an incoming edge. If there's zero pebbles in that reachable piece, then you merge. And you keep doing that. Those are all the things that are rigidly attached to you.

So I won't prove that. It's a little tricky to prove, but it's fairly easy to compute. And you could actually do it in the same amount of time. As you're building this thing up, you can do the color coding to figure out which things are in the same component, which things are in different components. Let's see. So another fun fact is this little part.

This says it's the 2k minus 3 property. The same algorithm you could use to check 5k minus 27, or whatever you want, for fixed numbers like two and three, the same pebble algorithm can detect that ak minus b property. And that's actually useful for a lot of different things. This is done originally by Lee and Streinu. Same lee, and Streinu we'll be seeing some of her results in a couple of classes as well.

For example-- so this is that-- 3D body and bar. So actually let me show you the kind of scenario we're thinking about. So I mentioned 3D. If we have vertices and bars between them, we don't know how to characterize generic rigidity. But a slightly different problem, which is 3D bodies, these are polyhedron. And they have bars connected between them. This we know how to solve in 3D and in any dimension.

And so this is a picture of not only having-- so their bodies can spin unlike vertices. Vertices can't rotate. Nothing happens when you rotate a vertex. Bodies have this extra degree of freedom. So there are two things we're allowing here, one is to add a hinge between two bodies and the other is to add bars connecting them from various random points. And it's generic, so these points will never coincide.

I think that's what makes this different from a vertex. Vertex bars always coincide. Here, they all just attach to generic points. So it turns out you can simulate a hinge by five bars. They're equivalent. So both of these can be captured by 6K minus 6 property. Turns out these things will be generically rigid if and only if you satisfy the 6K minus 6 property and you have exactly 6K minus 6 bars.

And this is proved by two or three people. I mean you need to add two results together, one from the '90s and one from much older, I think, in graph theory. But in particular, Tae was involved from 1984. And it's funny. The paper starts out with, if you ask any structural engineer if you have two bodies how many bars you have to add to rigidify them, they will say six. I'm like, OK, I didn't know that. But six is the right answer. And that's why the 6K minus 6 basically. And and we can use this pebble algorithm to solve that for free basically.

And this is actually implemented here in the same kind of software. I'll just run it quickly because it's a lot of pebbles. Every vertex has six pebbles. You just keep going. You need to cover every edge with how many pebbles? Before it was four, so now it should be 7, I believe, one more than the six here. So 6k minus 6.

Anyway, this graph, while it's not obvious, is minimally rigid in this world. And it corresponds exactly to the left example. It's hard to see because we've replaced the hinges here by six bars. So that's why we got all the six-- all the duplicated edges. Here, you see the duplicated are different from a rigidity standpoint because having three edges between the same pair of objects is no longer just the same as having one edge because they attach to different points. Each one can spin.

These are universal joints by the way. So these bars can rotate around the body. The body can spin. Lots of things can happen. But this is enough to rigidify. Is there a question? Cool. I'm not going to prove that. You can read the papers if you're interested. But it's interesting. Bodies and bars are so much easier than other things.

One other thing you can do is called angular rigidity. This is a fairly new result by same people. If you have a bunch of lines in 3D, and you have angular constraints. In this case, every triple of these lines has an angular constraint that it must be equal to alpha. That's what's drawn on this spherical picture. Over here, we have two bodies.

So here we have lines. Here, we have bodies. And there are three constraints that fix the angles between how the angles meet at these bodies. And they claim is that both of these are angularly rigid, meaning things can still slide up and down, but the angles are all fixed. And you can test this again.

This case actually turns out it's the 2k minus 3 property again, same as Laman. This one, I've forgotten. I have to check. I think it's 3k minus 3. Yeah, 3k minus 3. So cool things. One last question. Connected banana. So there's this three banana example-- or two banana example. I'm getting ahead of myself-- which was weird. You have this one banana on the left, one banana on the right.

And there's this implicit, implied hinge, as they call it, between the two points. The whole thing is flexible, but if you check it, it satisfies the Lama condition. It satisfies 3k minus 6, which is what it should be for 3D rigidity. This is a 3D. Now, this graph is-- seems trivial-- I mean it seems easy to figure out that this is flexible because there's a two cut. There are two vertices you can remove that disconnect the graph into two pieces.

So if you could just subdivide, do the left part separately from the right part, you should be able to figure this out. Unfortunately, this example can be made more connected, and that's what the question was. So this is an old example by Henry Crapo. A modification by Walter Whiteley is that if you add a single point here and attach it to those two points-- sorry, those three points-- this will be three connected, still be flexible, and still satisfy Laman because we added three edges and one vertex, so it still satisfies Laman.

So that sucks. We think, oh, maybe four connected. Well, you can make it four connected, too, by adding a triangle in the center and connecting these six, adding these 6 connections. Well, what about 5 connected? Well, five connected, you can also do. And this is an example called the banana spider. Although it seems a bit of a misnomer. It should be-- but I guess banana spiders are actual object-- actual species.

This really should be a banana insect because this guy has six legs, three on the left, three on the right. So if it is an octahedron in the center, and then you add these three connections, and you can actually prove that any graph you have, any example that's maybe only one or two or three connected, whatever, you can make it five connected by whenever you have two four connected components-- or five things components I guess-- just add a spider in the middle to bridge them, and it will be as flexible as before. And it will still satisfy Laman.

So five connectivity doesn't buy you anything unfortunately. I guess you could ask for six connectivity, but six connectivity, I think, is impossible because you only have 3n minus 6 edges. That's the limit. So sadly, connectivity is not the right-- doesn't buy us anything. And that's what we know about vertex and bar structures in 3D sadly. Tough open problem. Any questions? That's it for today.