ERIK DEMAINE: Today we continue our theme of dynamic graphs. This is the second of three lectures. And this will be the main lecture about upper bounds for general graphs.

Last class was about link cut trees. And we saw, essentially, how to solve a problem called dynamic connectivity, which is where you can insert into lead edges and you want to know what's connected to what. For trees, we solved it.

We're going to solve it again for trees in an easier way. Because we need a slightly simpler data structure we can augment in different ways, called Euler-tour trees. And then we'll solve trees, yet again, in a setting where you can only delete edges. Then we can get from log n time to constant time.

So we'll still talk about trees for quite awhile. But then we'll finally get to general graphs. And we'll do a log squared n solution for general graphs. So just another log factor more, you can generalize from trees to undirected graphs.

And then I'll give you a flavor of what's out there. Dynamic graphs is a big world. It has lots of results. And I'll give you some idea of what other problems have been studied in dynamic graphs, both undirected and directed. Though for directed graphs, things get pretty slow.

Yeah, next class will be about lower bounds, which will prove that this log stuff is actually necessary, in case you were wondering. Because a lot of the time, we've been able to get better than log in this class. And dynamic graphs, you can't. They're log lower bounds.

So let's start out with a definition of the problem. We'll be focusing, today, mostly on this dynamic connectivity problem. Maintain an undirected graph subject to [INAUDIBLE] insertion and deletion.

And you can also insert and delete degree-0 vertices. But we're not really going to worry about vertices. It's easy to add vertices that have no edges to them. Also easy to delete them. It'll be more about adding and removing edges incident to those vertices.

And then the query connectivity query is given two vertices. Is there a v to w path? So this is asking are two vertices, v and w, in the same connected component?

That's a strong query. In some sense, a weaker query is just to know globally, is the graph connected? That's another interpretation of dynamic connectivity. It turns out, both of these problems are interesting and useful for solving problems.

But they also turn out to be, more or less, equivalent. All the upper bounds we know for one apply to the other and vice versa. So it doesn't seem to make much difference which kind of query you want. So might as well take both. We'll focus on the v, w query.

OK. So that's the problem, in general. And I want to introduce some terminology, like when we had partially and full persistence, there's also a partial and full dynamicness. A fully dynamic data structure is one where you can do inserts and deletes of edges.

And a partially dynamic data structure is one where you can do inserts or deletes. So you can only do inserts, or you can only do deletes throughout the lifetime of the data structure. And these have specific names.

Because usually you use different data structures for just insertions or just deletions. Just insertions is called incremental. And just deletions is called decremental.

The idea of incremental algorithms is definitely not a new one. But in dynamic graphs, it always makes sense. So in general, a dynamic graph problem is defined usually by insertion and deletion of edges, and either fully dynamic or partially dynamic.

And then the query is what can vary. We're going to focus on connectivity queries. But you could do others.

So this explains, for decremental, we're going to solve trees in constant time. For fully dynamic, we're going to solve general graphs in log squared n time. I guess these three are sort of the meat of the lecture.

But before I go into that, I want to tell you where this dynamic connectivity results fit into the literature in general on dynamic connectivity. This is part of our survey, but just on dynamic connectivity. Well, for trees, this is the best we know.

This is the best, possible really. You can do log n in general, and then decremental. You can do constant.

We actually already know how to do trees with lint cut trees. It's not totally obvious, but you can use links to simulate edge insertions. And you can use cuts to simulate edge deletions.

It's not totally obvious because the link operation requires that one of the vertices is the root of its tree. So that's not so trivial. But we'll see an easy way to do that in Euler-tour trees later on.

So this we kind of already know how to do. But we're going to see another simpler way. The constant time amortized decremental we're going to do today. So we're also going to do this today.

Incremental, I don't think it's known, how to do constant time. But I'll tell you another result. You can get almost constant time.

Another special type of graph you could consider is a plane graph. This is a graph with planar embedding. So as you're inserting edges or adding vertices and whatnot, you say which face they live in. And so you have a fixed planar embedding. It's being constructed as you go along.

These are easy to maintain. You can get order log n. I guess, in some sense, that generalizes trees. Because they also have pretty obvious planar embedding. So that's something you can do, in log n.

But the big question is, for general graphs, can you do log n per operation? That's an open problem, probably the central open problem in dynamic connectivity. But we're not so far away from that.

As I said, you can get log squared. Let me say what the log squared result is. It's a little bit better than just log squared. You get log squared update, but query is actually sub logarithmic. You got a log n over log log n query.

And another result is you can do a little bit better than log squared. You can do log times log log n cube update. So roughly log n update, but there's some poly log log term.

And query slows down very slightly. It becomes log n over a log log log n. Well, it'll sort of makes sense why in a moment.

So the lowest update time known is this one, log times log log cubed. Big open question is whether you can get log for update, and any kind of reasonable query time. But ideally, log for both.

So that's for a fully dynamic. If I don't say any otherwise, fully dynamic is always the default.

Result, that's the case you usually care about. Cool.

But if you want to do incremental, you can achieve an alpha bound. Because incremental dynamic connectivity is really the union find problem. Union find, you have a bunch of sets of elements. You can take two of them and union them together.

Once they're unioned, they can never be split apart. And find is tell me what set I'm in. So if I do find on two vertices, I can find which connected components there and check whether they're the same. That will give me connectivity query.

Insertion corresponds to merging two connected components unless they were already together. And it's known you can get an alpha amortized bound. And that's optimal for that problem. So I guess that's actually a theta. So that's sort of well-known stuff. It's a complicated analysis, but already been done.

So in particular, that's, I believe, the best way we know how to solve incremental connectivity in trees. Though I'm curious whether you can do constant incremental. Decremental, there's essentially a log n solution. Not quite, but for dense graphs, there's a log n solution.

So I'm stating, instead of a bound per operation, this is a total bound. Let's say you start with an m-edge graph and you delete all the edges. Then you pay log n for each edge.

But you also have to pay this n poly log n cost. So if m is a bit bigger than n, then this dominates. And so it's log n per operation. But if you have a sparse graph, it's still poly log per operation.

So decremental, we can kind of get log for general graphs. Another goal you might have is what if I wanted to get worse-case bounds? Everything I've said so far is amortized.

If you want to get worse case, not much is known. Big open question is, can I get a poly log updating query. So it was actually a pretty big breakthrough to get poly log whatsoever.

This first result is from 2001 or so, after people have worked on dynamic graphs for probably a decade. So it took quite a while to find this any kind of poly log. That still hasn't been made worse case. So the best results are state of the art at the time, before the poly log came out, which is square root of n update, constant query.

There's one other result for worse case, which is if you want to do an incremental solution, so you're just inserting edges. Then there's a bound. It's actually a whole trade off between updates and queries. If you have updates to take order x time, then you get queries that take log n over log x time.

So log base x of n. And that's optimal. These are theta. There's matching lower bounds. So this is again, really union find. And so union find has been well studied, even in the worse-case setting. Can't do as well as alpha, basically.

OK. And then finally, I want to talk about lower bounds. So there are two complementary results. But maybe first I'll summarize by saying you need omega log n update or query. One of them has to be at least log n to do dynamic connectivity.

As you see, queries can be sublogarithmic. We don't know how to make updates sublogarithmic and get any reasonable query time. But one of them has to be at least log. And here's the specific theorems that imply that.

That's actually similar to this result, but not quite the same. Because there's a log on the left-hand side. And these results hold for amortized solutions. Whoops. Key word's missing here. This is an update bound. And this is a query bound.

So these are basically symmetric results. Says if you have super logarithmic update, so you have some x bigger than 1 times log n update, then you get log n over log x query. That's a lower bound.

If you have a super logarithmic query, so let's say it's x times larger than log n, then you need at least a log n over log x update time. So this result is the one of relevance. Because we don't know any sublogarithmic updates. But we do know sublogarithmic queries. So this is the regime that we have, essentially, matching upper bounds.

Let me show you. This result, this result, and this result, even, are, in some sense, tight against this trade-off, this update query trade-off. So this update is log n times log n. So x is log n. And over here we get log n over log log n. So that's tight against that.

Here, also, we have here x is log log n cubed. So you take a log of log log n cubed, you get log log log n. So that's why there's a log log log n down here. That's the best you could hope for, given that update time, that's the best query time you can get.

Same over here. Given an update time of root n, well, of course, the best you can hope for is constant query. But indeed, it works out. As you're taking log n divided it by 1/2 log n, you get constant.

So these results, they may sound suboptimal. But a certain sense, they're optimal. Of course, these are just single points on the trade-off curve. We still really care about the situation of-- where's the open problem here-- log n update and query, that would still be ideal.

But these are not as bad as they look, is the point. Of course, it'd be better if this wasn't a 3, it was only a 1. But they're still, in a certain sense, tight against this, the query bound that you get.

Another open problem here is whether this, the reverse situation, is ever relevant. Can you have a sublogarithmic update time and get any poly log query time? So this is another open problem. Little o of log n query. Sorry, little o of log n update, and let's say poly log query.

The lower bound kind of predicts that something like this ought to exist. But it may be it's just impossible. And there's a different lower bound that makes it impossible.

Some theta I wanted to add here. OK. Another fun thing about this lower bound, is that it holds even if your graphs are paths. So in particular, that tells us that this dynamic tree result that we covered last class, say, you need log n. If you're going to do both operations and log n, that's optimal.

Even if your trees are paths, which is like the really easy case. And we're going to prove this theorem next class. So if you want, that'll be our last class, our last lower bound.

That's next time. Tune in next time. But today we're going to focus on upper bounds, namely these two and this one. So let's do them.

The first one is Euler-tour trees, which is another way to do the log n bound. It's something we need. We'll need it for doing the log squared solution, for a reason we'll see in a moment.

So Euler-tour tress go back to 1995, Henzinger and King. They're simpler dynamic trees, simpler than link cut, even though they're newer. And the key difference compared to link cut trees is that they let you do stuff with subtrees of the tree instead of paths of the tree.

Link cut trees are great. We could compute the min or the max or the sum of a bunch of weights on any path that we cared about. But for various reasons, we want to do the same thing on subtrees. And this turns out to be easier.

So what do we do with Euler-tour trees? We take an Euler-tour. Remember Euler-tour from-- I can never remember-- lecture 15.

[LAUGHS]

I don't quite remember what that lecture was. When did we do Euler-tours? Well, doesn't matter. You remember the idea of an Euler-tour. It was just walk around the tree, do a depth for search, and keep track of every time you visit a node.

So you visit some nodes multiple times. In general, the degree of the node is the number of times you visit it. Every edge gets visited exactly twice.

So it's the linear number of visits total. I just want to take that linear order of visits, pull it out straight, store that in a balanced binary search tree. That's Euler-tour trees. They're very simple. Store the node visits by the Euler-tour, and a balanced binary search tree.

What's the order? We're storing them in order by the order that the Euler-tour visits, the visits. So this will be the left-most, the min in the tree. This will be the next and then next. So if you do an in-order traversal of the tree, you should get the visits in this order.

This is, of course, a way to balance a tree, in some sense. I drew a balanced tree as the thing we're trying to represent. But again, it could be a totally unbalanced thing. But when you do an Euler-tour, you get order n visits. And you throw them into a balanced binary search tree, it's balanced, of course.

It's a bit of a weird thing, because you're not really preserving the structure in an obvious way. But there's one more thing we store, which will let us do, essentially, whatever we want. Each node stores two pointers into this structure to the first and last visit.

So like this node has a pointer to its first visit and its last visit along this structure. It doesn't keep track of any middle visits. Because there could be a lot of those. And we just want to have a constant number of pointers. This is going to be a pointer machine data structure just like link cut trees.

Let's see. How do we solve our operations, inserting and deleting of edges and connectivity? Actually, I'll start with just phrasing it like link cut trees. We can do most of the link cut tree operations. Remember, there was a find root, link and cut were our main operations. Than there was aggregation. But that will have to change.

But if we want to do find root-- so we're given one of these balanced binary search trees. In it is the node v. Well, actually, it's a little weirder here.

But we have some node v in our tree. And really, we have our pointers into this. This is a balanced binary search tree here. We have pointers to the first and last visit in here.

I don't care. Just take any visit, anything in here, just say the first visit of v. Then walk up the tree. Then walk down the tree on the left spine. That is the min.

And that's going to be the first visit of the root node. And so this has a pointer, in turn, to the root node. Boom, you found the root of your tree.

Again, we're maintaining forest of trees. And so we just need to find which tree v is in, walk up, walk down, that's order log n, because this thing is balanced, and we found the root. So that's easy.

Turns out, the other operations are not too easy if you can split and concatenate your tree. So let's start with cut. We need a bigger diagram.

So here's v. Here's the parent of v. I'll call it w. Here's the edge. And our goal, and the cut operation is to delete that edge, separate this tree from that tree.

So what I need to do is isolate this subtree of v. But conveniently, if you think of the Euler-tour, the Euler-tour, it does some stuff here. Eventually it follows this edge. Visits v for the first time here. Then it visits all the stuff down here. Then it visits v for the last time. Then it follows this edge and then does other stuff.

So the v subtree is a contiguous interval of the Euler-tour. So we just cut it open, cut it apart. So we split the BST at the first and last visits to v, which is exactly what we know.

We have a pointer from v to its first and last visits in the Euler-tour. So split the tree there. Split the tree there.

You know split in a binary search tree. You're given a particular key value, let's say, x. And you split into everything less than x, and everything great than or equal to x, or something like that.

That's a split operation. It can be done in log n time, and pick your favorite balanced binary search tree. Red-black trees, AVL trees, they can all do this and maintain balance in these two things. Even though they might be very different sizes, they'll both be log height relative to their own size.

This is obviously not quite enough. Because what we've done is basically cut here and cut here. Now we have three trees. We have the one we want for v, v is subtree.

That will correspond to the Euler-tour of exactly that thing. So this is the represented tree I'm talking about. And this is the balanced binary search tree.

But the rest of the tree is in two parts. There's the part to the left of v, before we visited v, the part after we visited v. Those we just have to stick back together, so we concatenate them. Yeah.

So I'll call them the before-v tree and the after-v tree. There is actually a tiny thing that has to happen here, which I just remembered. Which is we need, somewhere, to delete one occurrence of w, of the parent of v.

Because it used to be we visit w, then we visit v, blah, blah, blah. Then we visit w again. If we cut out this part, we're going to have two visits to w in a row. We really just want one visit. So this is a minor thing, but we need to do a delete in there.

So each of these is log n time. So the overall time is log n. Easy. This is a cut-and-paste kind of argument, all pretty cheap. OK, now let's do link.

Concatenate is the reverse of split, by the way. You have two trees. All the things in the left are smaller than all things in the right. You just want to join them together like a very restrictive kind of merge.

So join, recall, is you have a node v. And you want to make it a new child of w. w might have other children. It lives in some bigger tree. We want to add v as a child of w. So we're assuming here v is a root, for now.

So what do we do? Sort of the same thing. We sort of know we want to put v in here somewhere. It actually doesn't really matter where we put it. We're not keeping track of the order of nodes in here.

So I'm going to simplify my life and say, let's put v at the end. I want to make v the last child of w, like this. So we can find the last child, in some sense. After we visit the last child, we come back and do the last visit to w.

So I'm going to look at the last visit of w, cut the tree open there, and stick this part in. Cut the Euler-tour in half there. So split. w is tree at w's last visit. Right?

And then we do omega concatenate. We're going to concatenate, basically, before w's last visit. We're going to concatenate a single occurrence of w. Because there's a new occurrence of w now. We do it before and after v.

So we add another w. Then we add v's Euler-tour, v's BST, and then we do after whatever used to be there after w's last visit. Let's say, after and including that w visit.

So just the symmetric of this, here we had to delete one occurrence of w. Here we have to add one back in. But you just cut, paste in the thing you need, and rejoin everything. So again, very easy. It's a constant number of log n time operations, if you can do split and concatenate in log n time.

So you see this, it's like wow, why do we spend so much time with link cut trees? This is really easy. But they have their different applications. If you want to compute aggregates on paths you need to use link cut tress. It's the best we know.

This structure will not do aggregates on paths. But it will do aggregates on subtrees. Because subtrees are represented by these intervals in the Euler-tour tree.

So if you have a weight on all the nodes in here, say, you can easily find this interval and do a range query there. So you take the min of all those things. If you have subtree mins, you can take the min of log n things and in log n time, compute the min in an interval, or min over a subtree rooted at a node, or max or sum, or whatever.

So here you can do subtree aggregation in log n time. And we're going to need that for the log squared solution. One other thing I wanted to maybe mention, here we did join. And join assumed that v was the root of its tree. What if v is not the root of its tree and we still want to insert an edge?

Because insert, when we're maintaining an undirected graph, there is no rootedness. That rootedness is just a tool. You could say at the first step here, just root the tree somewhere so that we can have a well-defined beginning and end to the Euler-tour.

Of course, Euler-tour in it's the most natural form, is actually a cycle. And it would only have one visit here. So you can think of it that way. I'd still like to have the two visits conceptually, because that makes even the subtree rooted at the root an interval between the beginning and the end.

But the nice thing of the cycle view is if I wanted to change who the root is, say I want this node to be the root, that's just a cyclic shift of everything. It just means I want to start the cycle here and end it here. So I'd basically duplicate this copy of the vertex, singlify this one.

There's only one copy. That's the constant number of inserts and deletes. And now rotate.

[LAUGHS]

What does it rotate mean? Well, before it was a sequential order, starting here, ending here. Now, I want it to be a sequential order, say, starting here, continuing this way, and then ending back here. So if you look at what that requires, maybe use another color, I want this part. That's an interval of the old tree, of the old sequence. And then I want this part.

So there's two pieces of the old tree. And I just want to change their order. It's a cyclic shift. Cyclic shift is just a cut and then a rejoin the other way.

So I can reroute a tree in log n time as well. So this is why, in this world, join is equivalent to insert. If I want to do a general insert, first I reroute v's tree. I do a cyclic shift with one split one concatenate, like this.

And I get v to be the root. Then I can do the join. So again, constant number of splits and concatenates.

So it's another operation we can do, in log in. Everything here is log n. Cool. Any questions about Euler-tour trees. That's all I'll say about them. We will next move on to next result.

AUDIENCE: Can link cut trees also do a reroute?

ERIK DEMAINE: Can link cut tress also do a reroute? I haven't thought about that. I would guess so.

[LAUGHS]

But it's going to require some more thought. So let's see, link cut tree, we can do a little aside here. Link cut tree, we do an axis on v. So now we get that v is up here, right at the root of the tree of oxx trees. Now we've got other trees of [INAUDIBLE] trees hanging off here.

But in particular, this thing represents a path. It represents the route to v path. It actually has no right child.

So it's a little bit emptier, like that. So now we want to flip things around and say v is the root. We basically want to reverse the order of all the nodes in this tree.

So reversing the order of nodes in a binary search tree, you can't regularly do. But if you augment your data structure to say, I could basically mark these nodes as all inverted, treat left as right and right as left. It's basically the dyslexic approach.

You have a bit saying from here down to here, mark everything as dyslexic. And so you invert left and right. So I think, with this appropriate augmentation, you can reroute the tree in log n time. But of course, there's some details to check.

It might even be that's in the original link cut tree paper. But I don't remember offhand. So I'm pretty sure it works, and therefore, gives you dynamic connectivity. But if it doesn't, Euler-tour trees do it too. So we're covered either way.

So that's Euler-tour trees. Next on our list is decremental connectivity in trees. So we're going to solve a weaker problem, which is just deletions only instead of inserts and deletes. We're going to solve it faster. Constant time per operation. This is just a fun little result.

It's fun because it uses some techniques we know. So bound is constant amortized, assuming all edges get deleted. We need to assume this, essentially, because we're amortizing against the future.

So we need to get all the way to the end. Overall, over n operations it's going to take order n time. This is, again, a series of refinements, or a bunch of pieces of the solution that get combined together, not a sequential algorithm.

So the first observation is, of course, we can do order log n. We know, now, maybe two ways to do it. But you can use Euler-tour trees and only do cuts, life is easy. Link cut trees, you could also just do cuts. You don't even need rerouting. Rerouting was only for joins.

So that may seem obvious. But how do we reduce the log n to a constant? In general, in this class?

AUDIENCE: Make up tables.

ERIK DEMAINE: Indirection, and maybe look up tables. Yep. How do we do indirection in a tree?

AUDIENCE: Leaf pruning?

ERIK DEMAINE: Leaf pruning, leaf trimming, whatever you want to call it. Should call it the Edward Scissorhands approach, or something. So leaf trimming, what I want to do is cut below maximally deep nodes that are [INAUDIBLE] log n. Say with greater than log n descendants.

So we have our top tree. And then hanging off some nodes here, we're going to have some bottom trees. These each have, at most, log n nodes. These have bigger than log n nodes below them. So we can charge every leaf in the structure to the log n nodes below it.

And so up here we have at most, sorry, n over log n leaves. Not nodes. Nodes could still be linear if the graph is a path. But at most, n over log n leaves. So we have, at most, n over log n branching nodes up here.

So what we need is a structure for dealing with long paths in the top, a structure for dealing with log n size things down here, and a structure for combining the paths together. And that's the easy part.

If we treat each long path as a single edge, basically, we look at the compressed top tree, in the sense of compressed tries. But it's now a tree instead of a try. I guess it's a try. Whatever

We look at the compressed tree up here. That will have size, n over log n. And then we can afford to use structure one.

Why? Because in some sense, there only be n over log n operations performed. So it's not a space issue there we're trying to save. It's a time issue. So a little different from the past.

There are only n over log n times that you can destroy a path up here. Because there's only n over log n paths. Each time you destroy a path, I do an operation up here, I pay log n.

But if the total number of operations n over log n, total cost is linear. So constant amortized against the future. So I'm going to use structure one on the compressed top tree. And what remains is what I do in the bottom trees? What I do on the paths? And then how do I combine all those results together?

Let's first talk about how to combine the results. There are a couple of different cases. It So we're doing a query on v, w, and we want to do connectivity query.

So it could be that v and w are in the same bottom tree. In that case, we just need to do a query within a bottom tree. So as long as we can solve the bottom tree case quickly, we're happy. Constant time, I guess. That will turn out to be pretty easy.

That's the easy case. Otherwise they could be in different bottom trees. So it could be v's down here and w's in some other bottom tree.

So then we have three queries we need to do. One is like this. One is like this. We need to test the single edge, but that's trivial. And then we need to do a query in the top. Can you get from here to here?

Now, these nodes are a little bit special, because they are leaves. They will be at the ends of paths.

So this is really just a query we can do using structure one. Because we take either an entire path or no path at all. So we can look at the compressed tree and that's enough, because the leaves exist on the compressed tree.

That's not always the case. A different situation is this. We might have, for example, v up here, w up here, and v and w are not down here. So these are irrelevant. We just want to know, in the top tree, can I get from v to w?

Now, this is a little more awkward because v is going to live on some path. w is going to live on some path. But it might be in the middle of a path.

And so there's now three queries we need to do. One is can I get to the top of my path? And then can I get from one path to the other?

So these are branching nodes here. And can I get from this branching node to that branching node? It might not always be going up.

Sometimes you have to go down. But point is, a constant number of queries to path structures, to compressed structures. This is, again, the one structure.

And a constant number of calls to a bottom structure will suffice to answer the query. There's a few more cases that I haven't drawn. If you could be half of this and half of this.

But it can all be done, as long as we can solve bottom and paths. So let's solve paths first, I think. Oh no, bottom trees, fine.

So part 3 of the solution is solve a bottom tree. Here we can do it in constant worse case. So this is essentially the lookup table thing, although it's a pretty simple lookup table.

What we do, bottom tree has only log n nodes. So we can represent 1 bit per node in a single word. And what we'll do, store bit vector of which edges have been deleted.

In what order? I don't really care. Just pick some fixed order on the edges down here, and store the bits in that order. And say every edge knows its sequence.

So you could do a depth first search at the beginning, to label the edges in some canonical order. And then every edge knows what its bit position is in the vector. So when I go to delete an edge, I just set that one bit using an or operation.

Cool. So I can delete edges. I can actually even insert edges down here that used to exist. I could undelete an edge. But that only works in the bottom structure.

So not generally useful. OK, so it's clear how to do an update, how to do a delete. I just mark a bit. How do I do a query?

I want to know, given two nodes, v and w, one of them could be the root of the tree. I want to know can I get from v to w? Or there's only a single path that might exist. So it's a matter of has any edge along this path been deleted?

So a general way to write that is I have w. I look at v's path to the root. I look at w's path to the root. At some point, we've reach the LCA.

What I want to know is, have any of these edges been deleted? If yes, I can't get there. If none of them have been deleted, I can get there. It's an if and only if.

I don't care about these edges up here above the LCA. Because I only need to go to the LCA and back down. So what do I do?

Each vertex stores a bit vector of its ancestors, of the ancestor edges. Well, ancestor edges is kind of weird. It stores a bit vector representing its path to the root.

That's this red stuff. I want to know, for vertex v, what are the edges along the path to the root of this bottom tree? Just preprocessed. You can build this thing as you do the depth first search and store it.

Again, it's one word per vertex. So I can fit it with constant space. It's constant space overhead.

And now, if I have two vertices, I take the XOR of those two paths. That will give me this part. And then I use that as a mask into the bit vector, say, are any of these 1?

How do I do that? I just mask. And if the result is the 0 word, then I know none of them are 1. Otherwise, I know some edge was deleted and I'm screwed.

So I take XOR of v's path and w's path. And then I mask this thing, which stores which edges were deleted. And then I check whether that word equals 0.

If it equals 0, yes, I can get there. If it doesn't equal 0, I can't get there because some edge was deleted. So very easy, because we can fit log n bits in a word. OK, that's the bottom tree structure. Next we need a path structure.

Here we can use due constant amortized with a similar trick. So here's our path. We're going to use, essentially, indirection again. Again, we know how to solve this in log n.

When we have a sequential thing, our usual way of indirection is to split into chunks. Each chunk here, I need to be about log n in size. So I'll just make it exactly log n in size. So there's n over log n chunks.

So if I store a summary vector, we can think of these as being 0, 1, 0, 1, whatever, where the 1's mean this edge has been deleted. 0 means it hasn't been deleted. All right, let me write n over log n chunks, each log n edges. And I'm going to store each chunk as a bit vector.

Bit vector has log n bits. w's at least log n. So I can do the same sort of things I could do before. If I delete an edge, I just set a 1 bit in the chunk vector.

Now, that's good for local queries. If I want to do a long-distance query, I need to basically summarize, are any of these edges deleted? If so, I'll put a 1 up here. Any of these edges deleted? If not, put a 0.

And now, I want to structure over-- this, you might call the summary vector, like in [INAUDIBLE]. Done that before with indirection as well. Now, this summary vector has size n over log n. So I can afford to use our log n solution that we started with.

So use 1 on summary vector of chunks. Because again, the first time I set one of the bits in here, I have to do an update up here. But only then.

Once it's been done once, I don't have to update again. So there will only be n over log n updates. So I can afford log n per update. It'll still be linear time total, constant per operation amortized. So this is my solution.

If I have a query, let's say I want to know between here and here, I first check, can I get to the right endpoint locally? Which is a query within the chunk. Which is again, I just mask out these bits.

I say and now do I have the 0 vector? In which case, nothing I care about has been deleted and I can get there? Or if I don't have the 0 vector, I can't get there.

I want to know, can I get to the left from here? Again it's a mask and a check against 0. And then I want to know, can I go over this interval of chunks?

And that, I can use the summary vector for. And again, that is a mask of a subinterval, and then checking whether it has 0. So I take the and of these three results, and I'm done. That gives me a query over a path.

So it's kind of fun. We use two levels of indirection. One to reduce the number of leaves. And then these were sort of trivial.

And then within the structure, we could only afford things on the non-branching part, so we had to deal with paths separately. And there we use another level of indirection. But in the end, we get rid of all of our logs. And it's constant amortized for deletions in a tree.

Questions about that? That was our second result. We have one more.

Last result we're going to talk about is log squared n update, log over log log query, for general graphs. Finally, we graduate from trees to general graphs.

This is a result by Holm, de Lichtenberg, and Thorup, 2001 is the journal version. So we want to solve dynamic connectivity. We want to understand, in general, when two vertices are in the same connected component or not. That sounds tricky.

We're going to do that in a pretty simple way at a high level. High level is, we want to store a spanning forest. You know what a spanning tree is.

Spanning forest, well, your graph might be disconnected. That's the whole point of the data structure. If it's not disconnected, you answer yes all the time.

When it's disconnected you say, OK, I'll have a spanning tree for this connected component, spanning tree for every connected component. Together we call that a spanning forest. That's the maximal connectivity you can get, but represented as a tree.

Now, we have a great way to represent trees, Euler-tour trees. And if you somehow connected together two components, that is an insertion of an edge in an Euler-tour structure. Great.

So we can maintain all these connected components and merge them if we have to, using an insert in Euler-tour structure. And to do a connectivity query-- I don't think I mentioned this-- you do find root on v and w, see whether they're the same root. Because root is a canonical name for the connected component.

You can solve connectivity using find root in constant time. Well, constant extra time, log for the find root. OK, so everything looks great. I can connectivity.

I can do insertion. What about deletion? If I delete an edge that is not in my spanning forest, I'm happy. I have exactly the same connectivity as before, as proved by the spanning forest.

The trouble is when I delete an edge that's in the spanning forest. Then it's like, uh, maybe. So here's spanning tree, whatever.

And now let's say I delete this edge here. Then there's a couple possibilities. It was a graph, it's not a tree. So there could be some other edge that connects those two trees. Then I have to find it.

Oh, I'm going to find it. That's going to be annoying. Or it could be there's no such edge. Then I'm fine. Then it's just a cut.

But distinguishing those two cases is going to be our challenge. And that's where we're going to lose another log factor, but only another log factor. To only lose another log factor, what we're going to do is not just store one spanning forest.

We will store the spanning forest, but then we're going to hierarchically decompose it and say, well, yeah, there's this big tree. But some of the edges I'm going to put in the next level down. Some I won't. So some subset of this forest will be at the next level down.

And there's going to be log n levels. That's where we lose our log factor. And the weird thing is, there's no real reason to put things down, except we'll use it as a charging scheme. We'll prove that an edge can only go down log n levels. And then it has to get deleted before it becomes relevant again. So it will let us charge only log n times per edge.

OK, that's about as good as I can give you a high-level overview. Now we have to see the details. It's kind of amazing that this works, but it does.

So we're going to talk about the level of an edge. This is not a definition, per se. But it's a change in quantity over time.

As you do edge deletions, some edges are going to decrease in level. They all start at log n. Log n is going to be the top level. That's the whole spanning forest. They will only decrease, so it's monotone. And they can never get below 0. So we start at log n. They could go down. And the lowest value is 0. Now, Gi is my graph. And G is going to be the subgraph of lower-level edges.

It's going to say, level less than or equal to i. So in particular, G log n is the whole graph. So we always include lower-level edges.

So level 0 is going to appear in all the Gi's. But if we look at G 0, it's only level 0 edges. G1 is level 0 and 1. And so this is sort of a hierarchical decomposition of the graph. We have fewer edges at the bottom levels.

And there's going to be two key invariants we have over these structures. In variant one it's is going to be every connected component of Gi is small. It's going to be size, at most, 2 to the i.

This is really what's going to let us charge against something. Whenever you go down a level, the max size of a connected component goes down by a factor of 2. So at level 0, all components have size 1.

There are no edges at level 0. So I kind of lied. I guess the lowest level is 1.

At level 1, you can have two vertices. So there can be isolated edges in Gi. But that's it. You can't have a path of length 2 in G1, and so on up the tree. At G log n, you can have the whole graph. Whole thing could be connected. But this will let us charge to something as we go down in the tree. As we go down in levels, I should say.

So next thing we need is a spanning forest for each Gi. Fi is going to be spanning forest of Gi. So it's going to maintain the connected components of Gi.

And we're going to store that using an Euler-tour tree. There's this issue of pluralization. I'll say trees. Because Fi is disconnected.

Each connected component you use, you store using an Euler-tour tree. Together, it's an Euler-tour forest, I suppose. So that way we can do a query. And so given two nodes, we can know whether they're in the same connected component in Fi, just by saying whether they live under the same root.

Well, in particular, that means that f log n, which is a spanning forest of G log n, which is everything, that will let me solve queries. This is the desired spanning forest that will let me ask connectivity queries.

So all this infrastructure is in order to support deletes efficiently. But queries are easy. We just look at the top forest. That's the one we want.

OK, now second invariant, this is where things get interesting. Variant 2. The forests have to nest. So F log n, of course, has the most edges. F0 is going to have the fewest edges. But I want them to be contained in this nested structure.

All this is saying is that there's really only one spanning forest, F log n. Fi is just F log n, but restricted to the edges of Gi. So really we're trying to represent this forest. But then, as we look at lower levels, we just forget about the higher-level edges.

Restrict to the lower edges of level less than or equal to i, that's our smaller force. This is, in some sense, the hierarchical decomposition of the forest. Because there's really only one forest. That would make our lives way easier.

Fun fact, is that that forest, then, is actually not just any spanning forest. It's a minimum spanning forest, with respect to level. You've probably heard of minimum spanning tree. Minimum spanning forest is just the analog for disconnected graphs.

So we're defining the weight of an edge to be it's level. And so F log n can't just be any spanning forest. It has to prefer lower-level edges. Otherwise, this nesting structure won't be true.

Now, that doesn't uniquely define the forest or anything. Because maybe all the levels are log n. And then every spanning forest is a minimum spanning forest. But it's a constraint on spanning forest. Cool Let's go over here.

Let me quickly say how to do an insert and how to do a query. These are really easy. Delete is really where the action is, but just to make sure we're on the same page, and to introduce a little bit of notation, and say a little bit about what we do. We are going to store incidence lists, which is for every vertex, we have a list of the incident edges in a linked list.

So in constant time, we can add our edge to v and w's incidence lists. That's not maintaining any order or anything special. Then we also set the level of the edge to the log n. That's what I said, every edge starts at level log n.

And then there's one more thing, which is if v and w are disconnected, and F log n, we can tell that because we can do a connectivity query on F log n. We have that as an Euler-tour tree. So we can see whether v and w are in different components. If they are, we have to merge them. So we merge them. We This is what you call an edge insertion.

So this is an Euler-tour tree insertion, that we know how to do in log n. We reroute and we do a join. So that's cheap. And that's it.

So insertion is easy to do in log n time, actually. Its deletion that's going to be painful. Actually, we're going to charge through the inserts. So it'll be end up being log squared amortized. But worst case, it's log n.

Great. I want to say a little bit about connectivity. Now we know how to solve connectivity already. We do find root on v and w and in log n time, we determine whether they're in the same component. But I actually claimed-- it's maybe erased by now. Yeah.

But I claim not to log n query. I claimed a log over log log query. Turns out, that's really easy to do. You just change the top level minimum spanning forest slightly.

So I want to make F log n equal to a B-tree. Or actually, it's a bunch of B-trees, one per connected component, of branching factor log n. I guess it's a B-tree, so I have to have some slop. Theta log n, let's say.

Usually we said with Euler-tour trees, it was a balanced binary search tree. I'm going to make this particular forest use a log n way, B-tree. This is going to slow down updates. But it actually speeds up queries.

So to do a find root is now a little bit easier. Find root should just be log base log n of n, which is log n over log log n. Because find root, you just need to go to your parent, to your parent, to your parent.

Go to the top. And then you have to go to the leftmost place. But it's easy, in a B-tree, to always go to the leftmost place.

If I had to do a search within the node, that would be annoying. But going to the leftmost, that's easy. So I only pay log over log log for query, so I got my desired query time.

And I claim the update is kind of OK. Because it's slowed down, but we're already going to pay log square eventually. And here we were doing log.

So we don't need to be that fast. Update is now going to be-- let's see-- the height of the tree times the branching factor. If we touch nodes in a root-to-leaf path, there's log over log log of them.

For each one, we have to rewrite the whole node. So we pay log. But this is less than log squared.

So this won't end up hurting us. Because we only do this at the top forest level. We don't do it at all the levels. If we did it at all the levels, we would lose another log factor, log over log log factor.

But here, we do this once at the top, pay log squared over log log. Then we have to update log n other levels, each paying log. So we're already paying log squared for the lower level. So if we increase the top level a bit, not a big deal. So that's how you improve connectivity queries slightly.

And now, finally, we get to deletes. This is the moment you've been waiting for. How do we do a delete in this structure? What are all these levels good for?

So we're deleting an edge e. Its endpoints are v and w. First thing we do is remove e from the incidence list. If every edge stores a pointer to where it lives in the incidence list, you can do that deletion in constant time. Great.

So as I said, if e is not in this forest-- what if it's in a lower level of forest? Oh, I see, right. I forgot.

In variant two, variant two says the forests are nested. So if it's in any forest, it's going to be in the last one. So if it's not in the last one, that means it's not in any of the forests. That means we're done. We do nothing. That's the easy case.

We didn't destroy any connectivity, because the forests represent maximal connectivity. They're spanning. But if it's in the forest, then something changed.

Then we need to determine, like in this picture, is there a replacement edge? Or is there no replacement edge? In which case, when there's no replacement edge, we basically don't do anything.

We have to do a bunch of deletes in the forests. But yeah, is that what I want to say next? Yes, we always do that.

We're going to delete e. We have to delete e. So we're going to recursively delete it from f sub e dot level up to F log n. e dot level is the earliest forest it appears in. And we have to delete it from all those.

Each of those is an Euler-tour tree deletion, or cut. And so each of them pays log n total cost log squared n. So this is where the log squared's coming from.

That's great. Now we've successfully deleted the edge. But now we need to know, is there a replacement? And at what level is there a replacement?

Now, we know by invariant 2, that there could be no replacement of lower level. The point is, this is a minimum spanning tree. So e was the lowest level edge that could connect the two sides, these two connected components of the forest.

So if there's any replacements at e's level or higher, there might not be a replacement at e dot level. But there might be a replacement at a higher level. We want to find the smallest level for which there is a replacement. That will preserve invariant 2, that we have a minimum spanning forest.

So that's what we're going to do, loop over the levels to try to find a replacement edge. So we're going to start at e dot level, loop potentially up to log n, call it a level i. Then I want to identify, at level i, the two sides of the edge. Let Tv and Tw be the trees of Fi, containing v and w, respectively.

We just deleted the edge connecting those two sides. So we know that they're in two different trees, Tv and Tw. And one of them is smaller than the other. I'd like to relabel v and w so that the size of Tv is less than or equal to the size of Tw, size, in terms of number of vertices from there in the tree. So swapped v and w if necessary.

Here's the fun thing. If we apply invariant 1, then we learn these sizes are not so big. Claim, at most, 2 to the i. As you realize it's almost 2 to the i, you have to imagine the moment before this deletion happened, before we deleted the edge.

Because at that point, Tv and Tw were actually one tree. They were connected by edge e. We've just deleted them.

But the moment before we deleted them, invariant 1 held. And so that was a connected component. It should have size, at most, 2 to the i.

Now, we split it. But that's all we've done. So the sum of those sizes has to be at most 2 to the i. Because that used to be a connected component. Cool

So that means that size of Tv is 1/2 that, at most 2 to i minus 1. Because Tv is less than or equal to Tw. So it's, at most, 1/2 of the total.

So Tv is even smaller. In particular, what that tells us-- now we have Tv and Tw as kind of separate components temporarily-- we don't really know that they're separate at level i. But we know at level i minus 1 they are separate.

At level i minus 1, there is no replacement edge, by the minimum spanning forest property. So what we could do, at this point, is take Tv, take all of the edges internal to Tv, and push them down to level i minus 1. We could afford that.

What do I mean by we could afford that? We wouldn't destroy invariant 1. Because we're taking 2 to the i minus 1 vertices, pushing all those edges down.

We don't have to push all of them. We could push some subset of the edges down. Whatever connectivity component we make at that level will be of size at most 2 to d, i minus 1, and so invariant 1 will be preserved.

So great. We can afford in this certain sense of "afford," to push all of Tv's edges to level i minus 1. Great

We don't actually need to do this. But what we're going to do is use it to pay for stuff. We have this scary goal, which is we want to find is there a replacement edge?

I don't know a good way to find a replacement edge, except to do, basically, a depth first search and look at all the edges. That's going to take a lot of time. But the good news is, if we search from Tv, every edge that's useless, we can just decrease its level.

And whenever we decrease the level of an edge, we basically get a free coin to continue. So you get a free life every time you push an edge down by one level. Because overall, number of pushes can be number insertions times log n. Because every edge can only be pushed long n times.

So whenever we push down an edge, we get a free bonus life, so we can keep doing our search. So here's how the search works. I'm going to say, for each-- this is a little bit tricky to implement. I'm going to first tell you what we want to do and why that's OK. And then we'll see how we actually do it.

So I want to search from every vertex in Fv. I want to look at all the outgoing edges from there of level i. Never mind how to find those edges. Just pretend you could find it in constant time per edge. It's going to be log n time per edge. But that's OK.

It's two cases. Either y is in Tw. Otherwise, y is going to be in Tv. Because we have Tv here. We have Tw.

We just deleted this edge, e. Sadly, we want to find a replacement edge. So if we look at all the edges coming out of a vertex in Tv, this is, I guess, x.

It could be it's an edge that stays within Tv. Or could be it's an edge that goes to Tw. There are no other options, because of our invariants.

If it's in Tv, oh, that's kind of annoying. Doesn't help us. But then we can just set the level of that edge, e-prime, to be i minus 1. Because we can afford to push all of these edges down by one level. So that pays for this round of the loop.

What about the other case? The other cases is the good case. That means we've found a replacement edge for e. If we have an edge from Tv to Tw. And at level i, we did these levels in order.

Did I say that? Yeah, we're doing levels in order. So if we can find this is the lowest level replacement edge, so we do it. We insert that edge into Fi using an Euler-tour tree insertion.

And then we're done. So we stop the algorithm. The moment we find a desired edge we're happy.

Now, maybe there are lots of edges. So we've got to stop when we find the first one. Then we've restored our minimum spanning forest property. We have maximal connectivity, all these great things.

As long as we don't find an edge, we can push things down a level and pay for this round. So it's constant time or log n time to do the last step. But all the other steps are paid for by the decrease in level.

So the claim is overall, we pay log squared n. Because we had to do log squared n for these deletions. Plus log n times the number of level decreases. That's for this step.

I claim that each round of this loop cost log n. So that's why we've get log n times number of level decreases. But number of level decreases is, at most, number of edge insertions times log n. Because each edge can only decrease in level log n times, by definition of levels.

And so we can charge to the insert operations at a factor of log n. So there's two log n's here. So inserts now cost log squared amortized as well. To deletes costs log squared amortized. And we get log squared overall.

One last issue though, which is how do we do this step? Actually, there's another step that's not so trivial, this one. We need to know the size of Tv and the size of Tw.

Well, that's easy. I just store subtree sizes at every node. So then I look at the root of the BST, and it tells me how big that tree is. And so I can see which is bigger and switch accordingly. But that's trivial.

This one requires a different augmentation for Euler-tour trees. And again, in both of these situations, we're using the fact that we can augment on subtrees, not on paths. That's why we need Euler-tour trees, not link cut trees.

So what we store in F-- why did I write Fv? I mean Tv here. So in Fi-- this is for each i, but at the forest at level i-- what I'm going to store is at every node in the Euler-tour tree, I want to know in the subtree rooted in that node-- so here's Euler-tour tree. Here's a node, v. This is a tree in Fi.

I want to know, in this subtree, in here, is there any node that has a level i edge incident to it? So I want to know, are there any level i edges? If there aren't any level i edges, then for the purposes of this search, I should just skip that subtree.

It's as if the subtree was erased. So I have some tree. It has height log n overall. I erase some of the subtrees according to this bit, which is an easy thing to keep track of. It's an augmentation, constant factor overhead.

Then I basically want to do an in-order traversal of this tree, but skipping the stuff below. So I mean, one way to think of it is you just repeatedly call successor, like successor in a binary search tree. But whenever I see that there's no level edges below me, I just skip.

So in log n time you can basically find the next place where there's an outgoing edge. I guess also, the incidence lists have to be stored separately for each level. So you can say, what are your outgoing level i edges from a vertex?

This lets you find the vertex quickly, but then you have to find the edges quickly. So the instance lists are not just stored as one single link lists, one link list per level. But that's easy to do.

So then from this, vertex we can find all the outgoing level i edges. And we can do this 4 loop efficiently. And whenever it's low, we get to charge.

In the remaining negative two minutes, let me tell you briefly about other problems that have been solved. So this finishes log squared n fully dynamic connectivity. I wasn't planning to spend much time on this anyway.

There are notes. If you're interested in the specific problems, I'll just tell you what the problems are. Generalization of connectivity is k connectivity. I don't just want to know, is there one path from v to w? I want to know are there k disjoint paths from v to w?

So they could be edge-disjoint or vertex-disjoint paths, might be your goal. In that case, poly log solutions are known for 2 connectivity edge or vertex. This is the result, actually, I used in my very first graph algorithms papers, where I first learned about dynamic graphs.

In log to the fourth n time, you can maintain 2 edge connectivity. It's open for k equals 3, even, whether you can achieve that bound. So kind of lame.

Another problem is minimum spanning forest. So you don't just want to maintain some spanning forest. Maybe you actually have weights on the edges and you want the minimum spanning for us. That it turns out, you can solve with a couple extra log factors. So in log to the fourth n time, with an extra log squared factor, you can reduce to the problem of dynamic connectivity. And so log to the forth overall, you can maintain minimum spanning forest.

Another problem is planarity testing. You want to insert or delete edges and, at all points, know whether your graph is planar. And that, there's no great results known for that. It's like n to the 2/3.

Directed graphs, things get really hard. There are two problems I've surveyed here. There's probably more. But the most fundamental questions are the analog of connectivity is what I call dynamic transitive closure, which means given two vertices, v and w, I want to know, is there a directed path from v to w instead of undirected?

At the moment, a bunch of results match a trade-off curve which should be update times query is roughly m times n, a number of edges times number of vertices. This is big. But there are a bunch of results that achieve this trade-off. I don't think I'm matching lower bound.

But it's also open, whether this entire trade-off curve could be achieved. There's various points on the curve where you pay square root of n for an update. Then you pay like, m times square root of n for the query or vice versa, stuff like that.

And then a harder version of this is you have weights on the edges. And now the query is not just find a path, find the shortest path. This is what I call dynamic all pairs shortest paths. This is even harder.

There are some results where you can do like, roughly, n squared time updates constant query. So it's maybe similar type of trade-off curves, not really known. When all the weights of the edges are 1, you just want to find the shortest path in terms of the number of edges, you can do not really that much better. Still, this is the product. But at least more of the trade curve has been solved.

So that's a quick overview of dynamic graphs, more generally. Dynamic graphs come up all over the place, obviously. You have network where the links are changing, or Facebook, you want to maintain connections but people friend and de-friend all the time. So you want to maintain whatever structure you care about. Anyway, that's dynamic graphs upper bounds. Next class will be lower bounds, the log n.