PROFESSOR: Well, welcome back to computational systems biology. We're back here today talking about genome assembly. How many people have ever assembled a genome before? In your spare time? Anybody done any genome assembly here? One person?

I think genome assembly is a fascinating topic. And as you know, it's at the bedrock of all modern biology. We rely upon genome references for almost everything in terms of studying evolution, looking at the structure of genes, regulation of genes, differences between individuals. So it's really a very fundamental concept.

And we're going to talk today about two different ways of assembling genomes. And I think one of the takeaway messages from today's lecture is going to be that genome assembly is more of an art, in some sense, than a science. And one has to always be a little bit suspicious of a genome assembly given what you're about to learn today.

And, of course, genome assembly is becoming even more complex because it used to be that assembling the human genome was the big task scientifically in front of the community. But now there are billions of genomes waiting to be sequenced-- all the individuals in the world and to try and interpret them. And now you can get your genome sequence for between $5,000 and $10,000. How many people here are tempted to get their genome sequenced?

OK, I see about five hands-- six hands. Great. So let's look at the science behind genome assembly. The basic concept is that we're going to collect some sequence reads from the genome. And we're going to assemble them know what are called contigs for contiguous segments. And these represent uninterrupted portions of the

genome that are completely covered by reads that we believe are contiguous.

These contigs then will be paired together in scaffolds. And scaffolds are like contigs except that there are missing parts between the contigs in a scaffold. We don't know what those parts are. But we're able to actually glue them together by using read pairs that allow us to jump over the missing parts because we have read both ends of a molecule. But we don't know what's in the middle.

And then oftentimes we had physical mapping technologies where we actually can go back and assign location scaffolds to physical locations on chromosomes by using PCR sequences like sequence tag sites that physically locate a particular sequence identity to a physical location on a particular chromosome. And that provides us with a total genome map.

So today we're going to be talking about how to go from a hard drive full sequence reads all the way down to a set of scaffolds that include assembled contigs. And the way to think about this once again is that we start with conceptually a single copy of the genome. We amplify this. And in order to sequence it on contemporary instruments, we have to fragment it.

Now for those of you who were in last Friday's recitation, you heard Heng Li talking about the idea that sequence reads are getting longer. In fact, sequence reads up to 10 to 15 kilobases are now possible. And sequence reads even longer than that are going to be possible, which will greatly simplify the assembly process. But for now we're talking about the challenge of assembling short reads-- say 100 base pair reads off of contemporary sequencing instruments.

So we take the fragmented reads and the notion is that we know that they're going to align up like a puzzle. And all we have to do is line the reads up to recover the read sequence at the bottom-- the original genome sequence. And I should add that many of the illustrations in today's lecture are from Ben Lagmi. He was kind enough to allow me to use them for today's talk.

So the goal is to come up with that red sequence at the bottom from the original set

of reads but, of course, the read set that we're talking about is perhaps 200 million reads or even a billion reads as we'll see. And so it's quite a tough task to put pieces together given that we really don't know where they came from. And we don't know where they align because we don't have the red part to guide us.

Now today we're going to be talking about what's called *de novo* assembly. That means starting from scratch. You hand me your set of reads for your favorite organism. And we're going to assemble it today. That's different than what's called reference-guided assembly because, for example, if you're going to re-sequence me or you, there is a reference human genome. And it would be a simple matter to take the reads from you or I and map them back onto the reference genome as a guide to trying to reassemble our genomes.

However, as you can tell, if there's a large structural variation between the reference genome and our genomes, that process can fail. So we're going to be talking today about *de novo* assembly. And in the process of *de novo* assembly, oftentimes we talk about coverage, which is on average how many sequencing bases do we have for every base of the genome. Here we have for this little illustrative example coverage of about 7x.

Now, at the origin of the Human Genome Project, some calculations were done about how much coverage was required to cover the human genome. And we talked last time about library complexity. This is a slightly different idea, which is we want to estimate the probability the base is uncovered. So if we have the genome size as G and the number of reads as N and L is the length of a read, then N times L is the total number bases that we have. And that divided by the genome is the average coverage of a base.

And probably the probability that a base is not covered is the probability we're going to observe zero reads to that base, which is e to the minus lambda, roughly speaking, if we use a Poisson approximation. And therefore, the number of uncovered bases it will have is going to be roughly G times e to the minus lambda.

The next calculations can be thought intuitively as the following way, which is if we

have N reads, if there's going to be a gap after a read, there has to be an uncovered base after it. And so the number of gaps we're going to have in our assembly is roughly N times e to the minus lambda.

So this is a back of the envelop calculation. And now if we take some of our 1,000 genomes data, which we previously used and asked how well this approximation works, we see something like this where the x-axis is the total number of reads and the genome coverage in bases is shown on the y-axis. And these are all different sequencing experiments.

So you can see there the roughly green outline, which follows the approximately what we saw before in this Lander-Waterman rule. Could somebody tell me what they think is going on with the red lines that actually don't match up with that green line? Anybody have any ideas about why we need more reads out of those libraries to get better coverage? Yes?

AUDIENCE: There is probably some bias when you're amplifying them?

PROFESSOR: Yeah, there's probably skew in the original libraries we talked about last time. In fact, we talked about last time why the Poisson was not a great approximation for looking at libraries. And in fact, we might want to fit something like a negative binomial in this particular case.

So we've got our read set. And we can also talk about coverage at a particular base, which is different than average coverage just to be clear that there are two different kinds of coverage that one can think about. Here we see coverage at T of level six. And the other thing that we need to be cognizant of is that there are two reasons that we might-- two common reasons why we might actually see reads that overlap but don't agree at all positions.

The obvious reason is that there's an error in one of the reads. We get quality scores and so forth. And that can help us decide which is the truth. But the other possibility is that as you know, you have one of each of your chromosomes from mom one from your dad. And there could be allelic differences between these

chromosomes.

So when we're doing assembly, oftentimes we'll find that these allelic differences are going to pop up in terms of non-concordance of our reads. And we'll have to ultimately decide if we want to make a single diploid approximation of a human genome or we want to attempt to assemble a diploid genome. And if we're going to do a diploid genome, then we have to be quite careful and use somewhat different assembly techniques.

But the common reference genome is haploid. It's only considering one chromosomal sequence. Is that clear to everybody? OK, great. So we're going to talk about two general approaches to assembly today. We're going to talk about overlap layout consensus assemblers as exemplified by a string graph assembler. And we're also going to talk about De Bruijn graph assemblers today.

Now, overlap consensus assemblers were the first ones that were used in the Human Genome Project because reads were longer back then. However, as the number of reads has increased, those assemblers are more difficult to utilize in part because of the need to find overlaps between reads, as we'll see in a moment.

Whereas to De Bruijn graph assemblers are somewhat more efficient. But they lose certain kinds of information. So let's begin with these overlap layout consensus assemblers. And we're going to talk about three steps to build contigs and the scaffolding step can be thought of a similar between either the overlap layout consensus assemblers or De Bruijn graph-based assemblers.

So we're going to first build an overlap graph. What's an overlap graph? The essential idea is that when we take our collection of reads, we look for overlaps between the suffix of one read and the prefix of another read. And if we think of all of our reads, we want to build a graph that describes all of such overlaps.

And just to be clear, I'm not going to be talking today about the reverse complement of these reads. Actual assemblers have to represent that. But it just duplicates all the nodes at edges. So we're going to try and keep things uncluttered by-- that's

OK. Thank you. We're going to try and keep things uncluttered by not considering those today.

Now, one of the challenges is how to construct those overlaps. And we're going to be talking about graphs a lot. So I thought it was worthwhile just to review terminology. We're going to represent overlap graphs as directed graphs, which consists of a set of vertices, which are the objects represented by the circles in the edges, which are the lines and a directed edge goes from one vertex to another.

And there's also an equivalent representation in notational form on the lower part of the right of the slide as well as a graphical representation. We're going to be using the graphical representations of these directed graphs today. So the overlap graph is simply a representation of the overlap between reads.

And we pick a minimum length of overlap at times. But for the next few slides, I'm simply going to represent each node as an individual read. And the edges will be annotated with the amount of overlap between the reads. So if I hand you a set of reads, all we need to do is to compute this overlap graph. We'll talk about how to do that in a moment.

And you'll see graphically then what comes out of the process of computing the overlap graph. Now, it's possible that overlap graphs are cyclic because there are circular chromosomes. And as we'll see, it's also possible to get a cyclic graph out of a linear chromosome if in fact there are repetitive structures in the chromosome that cause a graph to cycle back on itself.

So how to find overlaps in efficient time is a key problem. And that's one of the reasons that people have shied away from using these types of assemblers is because the cost of computing overlaps has been thought to be N-squared where N is the number reads because you have to compare all the reads to one another.

However, a really clever algorithm was devised that used the technology we talked about last time. You recall the idea of the FM index and Burroughs-Wheeler transforms allowed us to index a genome and then to look up reads in time

proportional to the length of the read.

So here's the essential idea. What we're going to do is we're going to take all of the reads that we collect. And we're going to index them. And we can do that roughly at N log N time. And after we've indexed all of the reads, then we can use that same index to find overlaps very, very efficiently.

And you can conceptualize this as simply looking at a read that you have in your hand and looking it up in the index. And you'll find all the places that the suffix or prefix of that read batches. And you can trace back till you find all the places it matches where they hit an end of a read. And those all correspond to edges in the graph.

And it turns out that this is so clever that it eliminates redundant edges. So, for example, if I have reads that look like this where I have read one overlaps with read two which overlaps with read three. And read one and read three also overlap. An unreduced graph would have a representation like this.

But it turns out that we don't have to do that because we can simply reduce our graph to this because we know that read one and read three. Actually, this is the graph that we would have that would be unreduced. We can reduce the graph to eliminate this transitive edge and simply represent it in this fashion. So when we use these indices, we eliminate these transitive edges as we'll see momentarily.

So here's an example graph. The sequence is shown on the bottom. The read lengths are of length seven bases. And we're going to consider all overlaps a minimum size three. And the edge label is the actual length of the overlap between the reads. And you can see that at the outset that these overlap graphs are not necessarily simple. That tracing a path of the graph that represents the original string is not completely and totally straightforward.

So we need to come up with a way to articulate our metrics for how to trace a path to the graph to reconstruct a genome. And that comes to the question of layout, which is how do we formulate the problem of tracing a path through an overlap

graph?

So we'll first start with the idea of the shortest common superstring. The shortest common superstring of a string S is the shortest string that contains all the strings in S as substrings for a particular length of substring. So, for example, if we didn't have the constraint of shortest, then just finding a string that contains all the substrings is easy. You just put them all together. But if we want the shortest, then we need to be more thoughtful in terms of the way that we compute this shortest common substring. And here is an example of the shortest common substring for the substrings that I have shown you up there.

So one way to think about the assembly problem is that we're trying to compute the shortest common substring of all the reads that we have. And that will be the most efficient representation of those reads in a linear sequence. Now, we can describe this problem in terms of an overlap graph.

And if you think about the way that we would solve this in overlap graph, in the shortest strings, we want the maximum amount of overlap. So we want to trace a path through the overlap graph that gives us the largest amount of overlap, which gives us the shortest string. Right? So if we simply negate the overlaps, we want to minimize the total cost of the graph.

Now, it turns out that this problem is known to be a very hard computational problem. It's in the class of something called NP-hard because it's known as the traveling salesman problem. And when you think about the fact that we're going to have hundreds of millions of reads, this is not really going to be tractable. If we got rid of the weights, and we simply wanted to find a path through the graph, that's called the Hamiltonian Path problem. That's also NP-complete.

So the shortest common substring is a way to think about assembling. But we can't really necessarily optimize metrics because it's going to be intractable. So think about ways of doing this that are greedier. So here's an example of how we would compute the shortest common substring starting with the first string. And each step along the way, is a concatenation of strings or a collapsing of strings that works

towards building the shortest common substring.

And we get the input string and the output string. So we could articulate our assembly problem as a greedy SCS algorithm to try and put all the reads together to come up with a superstring. And let me just describe to you this will give us an intuition into what goes wrong with assembly in a moment.

But we do know there are some bounds on this-- that if we actually did the greedy algorithm, then the assembly that we got would be only two and a half times longer than the true shortest common substring. That isn't really very much comfort to us. So we're going to have to come up with different, more heuristic ways of approaching the assembly problem.

Here is another example. Now, this is the one that I want to show you where we start with a string at the top where we're going to be looking for minimum overlaps of three and these are reads of six long. And when we do this greedy algorithm, we come up with a string, which is shorter than the original beginning string we started with.

Can somebody see what happened here? Why are we missing part of the original string? Yes?

AUDIENCE: The reads were short enough. And they repeated enough that we never found out that it was of the length that it actually was. And so we just kind of [INAUDIBLE] did it [INAUDIBLE].

PROFESSOR: So the point was that the reads were too short to be able to unambiguously identify the number of repeats of long that we had in the original sequence. That's absolutely correct. So we're not able to disambiguate what was going on. And perhaps if we went back to our graph formalism we could solve this problem, right? Because here we have our graph and the overlaps are written in on the edges of the number bases that each one of these reads overlaps. And all we need to do is to trace through this graph to find the original string.

So here is one tracing, which gives a total overlap of 39, which actually faithfully

reproduces the original string, right? However, that's not the best tracing. A better tracing through this graph or path through the graph would be this, which gives us more overlap and gives us a shorter string. But as we know, even though it's better according to this metric, it isn't really optimum because it gives us the wrong answer. It's better but wrong.

So we're going to have to take into account other things when we do our assembly and our tracing of this graph to be able to come up with the best possible assembly. So if we increase the read length as was pointed out to span appropriately, we will be able to reconstruct the original sequence. And the point of this example is that we need to consider this when we're thinking about recovering repeat structures in genomes.

So if we don't have long enough reads, in this case reads of length 8, we're not going to go to recover the original repeat structure. And if we look at this, repeats are really the bane of assemblers in some sense. And as you know, roughly 50% of the human genome is repetitive content. So we need to be very, very careful in terms of the way that we utilize reads to be able to recover the best approximation of our genome sequence.

So here's another example where we look at l is minimum over length and k is the length of the reads. And you can see the sequence that we're trying to recover-- It_was_the_best_of_times_it_was_the_worst_of_times-- and the output from our greedy SCS assembler. And as you can see, we need to get up to a read length of 13 characters for us to be able to properly assemble that original sentence.

So the essential message here is that unless you have reads that are long enough to span repeats, you're not going to go to recover the original sequence exactly. And this can be also thought of in the following example. Imagine you have repeats that are tandem repeats out at the end of a sequence. And we're using the English language here because it's easier to see than if I put up a bunch of genomic sequence. But, of course, the principles are the same.

You can see that unless we have reads that actually are anchored and unique

sequence and span out towards a repetitive sequence, we can't really tell how many times the word bells is repeated. Another possibility is that we can actually coming from both sides. And if we can anchor our reads and unique sequence on both the left and the right side of a repetitive element, then we can figure out how many copies of something like bells is present.

But in the absence of that, we really can't do it. In fact, we wind up with a structure looks like this. We wind up with-- there it is-- a structure where we have-- let's just say that there are four different stretches of genome in disparate parts of chromosomes and we repeat sequence in the middle. The blue parts of the chromosomes are unique sequence. And the red parts are repetitive sequences.

What will happen is that if the reads aren't long enough, we'll be able to find out in each one of the four locations that we've gone from unique sequence to repeat sequence. And then we will get lost in the middle of this identical repeated sequence. And then on the right-hand side we'll once again transition back from repeated sequence to unique sequence. But we won't know how to put things together in the middle. Right? We won't be able to figure out what the path is through these repetitive elements.

So that's the essential point I'd like to make about repeats. And we can now turn to the question of layout and how to process an overlap graph towards making contigs. This is the actual layout graph. When we think about that sentence up there. And we say the minimum over that length is four characters. And we have seven-character reads out of the sequence. You can see it's a pretty messy graph.

If we clean up the graph by removing the redundant edges, the edges like this that span over reads and are implied by other reads, we can remove edges that are transitive over one reads or two reads. Now, my presentation is going to talk about how to remove these edges. However, as I said at the outset, if you use the algorithm by Simpson *et al.,* you actually don't generate these transitive edges in the first place.

But assuming that you didn't use an algorithm and you did generate them, you want

to get rid of these transitive edges like so. And it starts getting somewhat simpler as you begin simplifying the graph, removing these transitive edges. And then we can remove edges that skip two nodes. So here's what happens after you remove the single transitive edges in this graph. Yes?

AUDIENCE: So it seems that the transitive and verbal edges gave us a little bit more information about the genome. Do we lose some useful ordering principles by--

PROFESSOR: They provide redundant information. They don't really provide any additional information. It's the same linear sequence that's implied by those edges. Any other questions?

So we can then remove edges that span two nodes. And we get an even simpler graph like this. Now this is beginning to look more tractable because we can look at this and we can output contigs that correspond to linear portions of the graph, which should be linear sequence. And when we do that what we wind up with are two contigs. And there's just a bit of problem in the middle, which is that we're unable to resolve the bit in the middle and as a consequence, we know that that is the number of terms that are in that original sentence because we didn't have a read long enough to be able to resolve that.

The other problem that we can have in doing this kind of layout is that when there are portions of the genome that occur or sequences in the genome that occur multiple times, when we actually do this layout, we may find that the portions of the genome that occur in two disparate locations line up with one another. And it may be that as you exit the portion that's shared you get a mismatched base.

So that mismatch could be because you have disparate parts of the genome that actually have very similar sequence. Or it could be that you had a read error at the end of your read. And it's difficult to tell the two apart except by the amount of coverage that you have. We'll talk about how to prune graphs like this in a few moments.

But in any event, assuming that we have pruned the graph, we have done our

overlap. We've done our layout. We've found our paths to the graph for our contigs. And then what we find is that for each contig, we have many reads. And we're going to take those reads. And we're going to look at them. And as you recall, we could either have errors causing disagreement among the reads.

We could have allelic differences between mom and dad causing those errors, well, not really errors-- differences. And then we can take a consensus to come up with what the haploid genome is. So that's the essential idea of a overlap layout consensus assembler. We compute the overlap graph. During the layout phase we actually simplify the graph. And we find pass through it. And during the consensus phase, we take our reads, and we build a consensus sequence of the genome.

And as I said, this graph building can be slow. Although, we'll talk about how slow it is here in just a moment. And the challenge is that modern sequencing data sets are hundreds of millions of reads. So let's talk about a contemporary overlap-based assembler-- something called the stream graph assembler, which is done over at the Sanger in the UK. And there are three separate steps it goes through.

The first step is it tries to correct reads. And the way it does this is it actually looks at all the k-mers that occur in reads-- it tries to find sequences that are very, very rare and find sequences that are nearby in sequence base that aren't as rare. And it can correct bases that it believes are sequencing errors.

The next step is assembly once it has taken all these reads and corrected them. It indexes all the reads as I suggested earlier using an FM index. And then it can find the overlap from that FM index directly. And part of the assembly process is throwing away duplicate reads and throwing away reads that have low quality scores.

So that's the filtering step. It then has the set of contigs that it has generated. And it does something quite interesting to find the scaffolds is that it takes the contigs it's assembled in terms of linear sequence. And it completely re-indexes them once again using an FM index.

And then it takes all the reads that you started with. And it maps them back onto the contigs. And by mapping the paired reads back on to the contigs, it can actually figure out what contigs should be formed into scaffolds where there are holes that are breached by these longer reads. So it's using the FM indexed both for correction to find out nearby k-mers for assembly to find overlaps and for scaffolding to put things together. And it does its indexing three different times.

And just to give you an idea of how long it takes for a human-sized genome, it's actually quite expensive in terms of CPU time. It takes many days have elapsed time to assemble an entire human genome right now. And it's thousands of CPU hours to actually put a genome together starting from scratch. OK, so that's the essential idea of an overlap-based assembler. Are there any questions at all about overlap-based assemblers? Yeah?

AUDIENCE: So in the case of an error , it's obvious how you would call that. But in an allelic difference, hypothetically, there would be 50% of the reads would have one and 50% of the reads would have another.

PROFESSOR: That's correct.

AUDIENCE: So in that case does it assemble-- do you just bias towards whichever ones weren't easily amplified? Or do you assemble two sequences?

PROFESSOR: Most assemblers produce a single sequence. And I don't know how SGA decides between the different alleles because I don't recall what the paper said they did. But they have to essentially flip a coin to come up with a haploid sequence. Yes?

AUDIENCE: You said there was three different times that you index. What are the three?

PROFESSOR: Yeah, the question was I said there are three different they indexed. They indexed at the outset to find errors. They indexed the second time to do the overlap computation. And they indexed the third time to realign all the original reads to the contigs they have to figure out which contigs to put together into scaffolds. Right?

But they have this essential foundational platform, which is the FM index. And so

they use that over and over again to be able to do the assembly. These are all great questions. All right, any other questions about overlap-based assemblers. And you can see that if you think about how much coverage they get out of an assembler like this, it's actually, we'll compare all the assemblers at the very end.

But if you look at the number of bases of autosomes and the X chromosome covered by an assembly, you can consider that as a function of the minimum alignment length to a referenced genome. And as the minimum alignment length goes up, that means you have to match longer and longer portions of the reference genome for your assembly contig to count. You can see that the number of bases dropped somewhat. In here they're showing that they do better than another assembler called SOAPdenovo.

But they do get a fairly good coverage. On the other hand, they don't get coverage anywhere near as good as Lander-Waterman might suggest because the coverage should suggest that the probability of uncovered base using Lander-Waterman would be roughly e to the minus 40th-- something like that. And e to the minus 40th is like 4 times 10 to the minus 18. So they're not anywhere near what we would think the Lander-Waterman bound would be for assembly.

So we've talked about these overlap-based assemblers. Now I'm going to turn to De Bruijn graph assemblers. How many people have heard of De Bruijn graphs before? Anybody? One person? So before we talk about De Bruijn graphs themselves, let's just talk terminology. So when I'm using terms we're all on the same page where we were talking about k-mers where the word mer is from the Greek "part."

And we talk about 4-mers of an original sequence as a sequence that's four bases long. And we can think about all of the 3-mers of an original sequence. So we talk a lot about k-mers. And a k minus 1-mer is a substring of length k minus 1 obviously from a k-mer. So if we think about the collection of reads-- here these are our super-simple economy sequencers producing reads of only length three, which is pretty desperate. But at any rate we'll go with that for the time being.

And we think about each one of these reads as having a left k minus 1-mer and a

right k minus 1-mer. We split them into two halves that way. And we're going to build a graph that is as follows. We're going to take all of the k minus 1-mers-- in this case the 2-mers. And for each read, we're going to draw an edge between its left 2-mer and its right 2-mer.

OK, once again, for each read, these sort of anemic, three-base-pair reads, we're going to draw an edge between its left 2-mer and its right 2-mer. And they overlap in one base. So all of the graphs that are De Bruijn graphs, the edges represent an overlap of one base. OK? So if you look at the graph at the bottom, that represents the overlaps present in the original sequence. You note that we have AA as one of the 2-mers. And its left half and right half obviously overlap by one base.

The triple-A read has AA as its left read and AA as a right read-- thay overlap at one base. And that's why we have that circular edge from A to itself. And the next edge from AA to AB comes from the next read-- the AAB read. So each edge then represents an overlap of one base. And therefore, each edge represents a unique k-mer sequence.

So the way to think about this graph is it that all of the edges represent the original reads. And we have represented the k minus 1 words as the nodes. OK? So we can take this graph then and generalize this idea. And if we look at how the graph changes as we add more structure, here you see that we've added an extra b. And we get another edge in the graph back to the same node.

So when we're building these graphs, if possible, we reuse a node that already exists. Now the way to think about coming back to the original sequence is finding a path through this graph and emitting sequence as we trace the path. And we would like to have a path that traverses all of the nodes.

And so we have some definitions here, which is that a node is balanced if its indegree equals it's outdegree. And you can see that not all the nodes are balanced down the graph of the lower, right-hand corner. And it's connected if all the components or nodes can be reached. And a Eulerian walk visit each edge exactly once, which is what we would like to actually take a De Bruijn graph and emit a

genome sequence.

Now, not all graphs have these walks. And graphs do our Eulerian. And we won't distinguish different types of these graphs. And if a graph has two semi-balanced nodes and all the rest of the nodes are balanced, then it will have a walk through it. So if we think about our original graph, there are two arguments for it having such a walk. The first argument is that we show the walk. And the second is that we have two semi-balanced nodes and the rest of the nodes are balanced.

So the reason that we care about this is that we want to study cases where this goes wrong. So to build a De Bruijn graph of a genome, we're going to take our original sequence reads. And we're going to take all the k-mers that occur in those reads. And we're going to add edges to a De Bruijn graph based upon those k-mers.

So if we have a read like this, and we consider a k-mer in the read, we're going to add an edge in the graph between the left k minus 1-mer and the right k minus 1-mer. And we'll do that for every single k-mer in the read. Now note what this does is it destroys some information. It destroys information about the ordering of certain of the k-mers in this read just destroying their read contiguity in order to make some simplifying assumptions to represent the sequence ordering of these k minus 1-mers in the graph. So we build the graph in this way and if I were to build the graph like this, what is the minimum sequence overlap for two reads to actually share an edge in the resulting graph? Can anybody see how long the sequence must be in the second read for it to actually overlap at edge with the first read?

Well, if this second read also has a k-mer, right? It's going to produce another structure just like this one if these two do overlap. And thus the edge produced by this read and the edge by this read will overlap like this. And thus all of the nodes that came from this part of read one will feed into this graph. And then all the nodes to come out of this k-mer from the purple read will come out of it like so, right?

And thus when we're tracing the graph, the idea is that the graph will be connected. And we'll be able to come between these reads and reconstruct the sequence that

was suggested by the overlap. The thing, however, you should note in this-- yes, question?

AUDIENCE: So you're picking two k minus 1 reads there-- are those from different reads? Or from the white read?

PROFESSOR: No, it's from the white read. These are the 2k minus 1-mers that came out of this read. So they actually overlap.

AUDIENCE: Yeah, but then you were talking about how the one was purple in that case.

PROFESSOR: Right, well, this is the same sequence let's say. This is the same, exact sequence down here. So if it's the same, exact sequence, it will have the same k minus 1-mers. And when we build the graph if a node already exists, we reuse it. And thus if we reuse the nodes that were created when we built the graph nodes and edges for the white read, then when the purple read comes along, we're going to put another edge here between these two k minus 1-mers because they are contained here as well. So these are identical sequences to this because these two reads overlap. And this part is the same sequence as that part.

AUDIENCE: Yeah, so why do you need k minus 1-mers if you have overlapped k?

PROFESSOR: Because the way we're finding these overlaps is through the graph. And we're not indexing things of size k, right? We're indexing things of size k minus 1. In each edge represents a sequence of length k because we know this sequence and this sequence are overlapped by one base.

So when we find an edge that's the same between the white and the purple read, we know that they're overlapping by k bases. Is that making sense to you?

AUDIENCE: No.

PROFESSOR: No, OK, so let's try it again.

AUDIENCE: You can keep going.

PROFESSOR: No, it's OK. Let's just start with the purple read to start for a moment because I think if you have a question, other people may have a question. So we have this sequence, which is this sequence right here, right? And then we have this sequence, which is the sequence right here. They overlap by one base. And so we put an edge between them like this in the graph. OK?

AUDIENCE: Don't they overlap by more than one base? They can only contain one base from each k-mer.

PROFESSOR: I'm sorry. That's what I meant. Yeah. And then the same thing is true down here. And so we will find this k minus 1-mer and this k minus 1-mer. And then they overlap. For genome assembly, we record the forward and reverse complement reads in twin nodes. And we're not going to show those because it just complicates our graphs without really adding any illustrative power.

And we always choose k to be odd so that a node can't be its own reversed complement. And here is the graph growing if we think about k equals 5. So we have reads of length five. And we are adding sequences to the graph. And you note that the graph is acyclic until we get to the repeated sequence.

And we get to the second long the sequence comes back around begins a looping back on itself. And if we consider the last part of this De Bruijn graph construction, then we wind up with the finished graph on the right-hand side. And you can see the multiplicity of the edges correspond to the number of times the long is repeated in this graph.

So once again, repeats are causing the circular structure, which only could be resolved if we had sufficiently long reads, which we don't have in this particular case. However, if we consider perfect sequencing we always have a path to the graph. And the reason is that the leftmost part of the genome, so to speak, is going to be semi-balanced. And the rightmost part is going to be semi-balanced. And all the parts in between are going to be balanced.

So the k minus 1-mer on the very left end is semi-balanced and the k minus 1-mer

on the right is semi-balanced. And all the nodes in between are balanced. Now, this does not allow for errors of course. And we talk about following this Eulerian walk to find the original sequence. But the question we can ask ourselves is whether or not this walk always really corresponds to the original genome sequence.

It turns out I can show you this example, which is we have this graph for this sequence. And there are two different walks through this graph. And the two different walks produced two different sequences. And they depend upon which way you start walking from the node AB.

So once again, here we have seen that even when we have a path to the graph, the path may not be unique. It may not be able to generate the original sequence that we started with. So the other problem we can have when we are building a graph like this is that gaps in coverage can create holes in the graph.

So if we omit certain of our reads, we'll come up with a graph that is broken into two parts. And this corresponds to the idea that we're going to create two different contigs that are contiguous sequence but will be unable to fill in the middle part. OK?

So we also can have differences in coverage of a graph when we have extra reads at particular locations in the genome. And that causes the degrees on the individual nodes to vary and causes us to not be able to rely upon the indegree and outdegree as an absolute metric for how to trace a path through the graph.

And the other thing is that if you have differences between the chromosomes, which we talked about last time in our overlap layout consensus assembler, it also can cause graphs to split apart and to have subgraphs that correspond to one allele versus the other allele, which is present perhaps in the main graph.

All right, so it's actually the case that these graphs are attractive for a very important reason, which is there extraordinarily efficient to build. That is in order to build a graph like this, you need to take each one of these k minus 1-mers and actually find the node, which you can do by hashing and then put the edges into the graph. And

so you find that you need to put in an edge and two nodes for each k-mer. And if you have a hash map that encoded these nodes and edges, it's constant time work. So you wind up with a graph which costs order of the number of reads to build.

So it's a linear time graph construction problem. Recall that our last overlap construction, we thought we could get down to N log N. And here is an example of sub-setting part of the lambda phage genome using a De Bruijn graph assembler. And you can see that roughly the time required to assemble parts of the genome is linear in the amount of genome sequence that you give it.

So these assemblers were favored early on in the days of short-read assembly in part because they were so efficient. And typically in some of the projects, you have very high coverage. And so you wind up with graphs that actually have a huge number of edges between nodes. And this can be summarised in terms of a graph that simply annotates the edges with the number of instances.

And so you have a weighted graph on the right-hand side, which is easier in some sense to trace because we can now begin to eliminate low-coverage edges as potential anomalies. But the essential idea is to trace these graphs to produce the ultimate genome sequence. And in order to do so, we may need to do some error correction.

So we talked earlier about the idea that if we have an error, we're going to actually produce a portion of the graph that hangs off into outer space. And we can cut these dead-end tips of the graph off if they are low coverage because they presumably correspond to errors.

If we get an error in the middle of a read, we can wind up with a so-called bubble in the graph, which once again is low coverage. And we can get rid of these bubbles in a similar fashion. And it's also possible to get chimeric edges of the graph. And those can be caused by errors as well. And we can clip those edges.

So there are different kinds of error correction we can do in the graph. These are all quite heuristic. Each assembler has its own set of heuristics for how to deal with

graph anomalies and how to eliminate edges in the graph to permit assembly. But these are getting rid of dead-end tips and popping bubbles and getting rid of chimeric edges are important things to consider for any assembler.

So the limitations of these graphs are the idea that we're immediately splitting these reads into this k-mer representation, which is destroying information. And in order to overcome this, one of the things that people have done in these De Bruijn graph assemblers is to take the original reads and to map them back on to the graph.

So when you're attempting to trace the path through the graph, what you do is you take the original reads. You thread them through the graph. And you know that the original read represents contiguous genome sequence. So it provides you with a path through the graph that you know is good.

People have been doing this in part because they didn't want to go to the full overlap graph implementation because of the cost. But I think that these overlap graph implementations now are sufficiently sophisticated that I personally would use them instead of a De Bruijn graph assembler. And so the trade off really centers around speed and space versus accuracy.

So we can look at some example assemblers and look at their performance. But before I do that and we leave De Bruijn graphs, are there any other questions about De Bruijin graph assemblers?

AUDIENCE: I have one.

PROFESSOR: Yeah, question.

AUDIENCE: How long is k typically?

PROFESSOR: We're going to talk about that. The k typically is somewhere around 60-- something like that-- Somewhere in that neighborhood. It's actually-- it has to be odd, right? So 61, 57-- something like that. Good question. Any other questions about De Bruijin graph assemblers?

So once again returning to over our architecture, we have these reads. We need to

produce contigs. In the case of overlap graphs, we're going to trace the overlap graphs. In the case of De Bruijn graphs, we're going to trace the De Bruijn graph.

For scaffolding, we can use the read pairs to put scaffolds back together again. And here is some comparison of the performance of these various assemblers. So the first assembler-- SGA-- is an overlap layout consensus-style assembler. Velvet/Abyss and SOAPdenovo are all De Bruijn, graph-based assemblers. So these are all contemporary assemblers that people use for assembling genomes.

An important metric for assemblers is something called N50, which is the size of a contig or scaffold where at that length or larger 50% of the bases are present in scaffolds of that length. So, for example, for SGA, they say that scaffold N50 size is 26.3 kilobases, which means that in scaffolds of length 26.3 kilobases or larger, half of the bases of the assembly lie.

So the larger the N50 is, the larger the scaffolds are that cover things. And you want larger and larger scaffolds or contigs so that you have fewer gaps in your assembly. So the N50 number is a principle comparison metric when one is thinking about assemblers.

So in this particular case, for SGA the overlap metric was that the reads had to overlap by at least 75 bases or more. And these were 100-base pair reads. You can see the details on the read data on the bottom line there. So as long as the reads overlap by 75 bases, they were put together in the graph.

And the De Bruijn graph assemblers each had their own optimum number for k. And the way that you tune these parameters is you run the assembler on a range of k values. And you see which k value produced the assembly with the highest N50. And you pick that k.

Can anybody think of a reason why it is that although these are all roughly in the same ballpark, different assemblers might have different k values given that the underlying technology is quite similar? Any guesses about what is going on here?

Well, we know that the differences in the assemblers really are rooted in the way that they are processing the graphs and the way that they are simplifying them. And therefore, one has to imagine that the differences lie in the post-processing of the graph once it's built and that certain assemblers like larger k values. Whereas other ones can tolerate smaller k values.

And you can see if we look at the running statistics for these, that the performance of SGA if you look at the reference bases covered by contigs greater than one kilobase is roughly comparable to all the other assemblers. But its mismatch performance is much better. That is the other assemblers are producing-- well, I take it back except for SOAPdenovo. But it does quite a good job at correcting reads in coming up with the correct sequence.

The last lines however tell the story about running time, which is that the overlap consensus assembler is taking 41 hours of CPU time for C. elegans genome assembly. Whereas the other assemblers, the De Bruijn assembler are running much faster.

So the thing that I wanted to emphasize today was that once you have the final graph whether it be an overlap graph or a De Bruijn graph, which represents possible ways of putting back together again the jigsaw puzzle, it still is an art to be able to build an assembler that uses appropriate heuristics to trace the graph to come up with a genome sequence.

And I think another lesson is that repeats are very problematic. With short reads, we really cannot resolve repeats exactly. As a consequence, when we think about any reference genome that we're dealing with, if we consider the size of the reads that were used to assemble that genome, then we need to be mindful of what that tells us about whether or not the repeat structure that we're observing in the genome is really an accurate rendition of what's going on in the genome itself.

And finally, I think that we've talked today about the problem of assembling genomes from a set of reads that represent a uniform, single individual albeit with possibilities of differences of alleles between mom and dad in a diploid organism.

However, environmental sequencing where one takes up sea water or other samples and sequences all the organisms in it and then attempts to assemble those organisms *de novo* admits the possibility that there are many different genomes that you're considering.

And that, of course, creates a whole new set of research problems, which I think are unsolved in part because of the read links that we're currently dealing with. Are there any final questions about assembly? OK, great. Well, we will see you then on Thursday where we will talk about ChIP-seq and IDR analysis. Until then, have a great Wednesday. Thank you very much.