Next Generation Sequencing Technologies: Data Analysis and Applications

De Bruijn Graph (DBG) based assembly

Dr. Riddhiman Dhar, Department of Biotechnology

Indian Institute of Technology Kharagpur

Good day, everyone. Welcome to the course on Next Generation Sequencing Technologies, Data Analysis, and Applications. In the last few classes, we have discussed two approaches to genome assembly. The first one was the Shortest Common Superstring approach, or SCS approach. We have also talked about a greedy version of that. And then in the last class, we talked about an approach called the Overlap Layout Consensus Approach, or OLC Approach.

In this class, we will introduce a new approach called De Bruijn graph-based assignments, which we will discuss in comparison with the other approaches that we have discussed so far. So this is the agenda for this class. We will be talking about De Bruijn graph-based assembly. And here are the keywords for this class.

The first one is a directed multigraph, and the other is an Eulerian path. So just to summarize what we have seen in the overlay layout consensus approach So there are three steps. The first step is building the overlap graph. Then the second step is doing the layout, where we remove these inferred edges that can be inferred from other edges of the network.

And then the third approach is consensus calling, where we determine the sequence of the reference genome. And we have talked about how repeats can impact this OLC method, but what I would like to encourage you to do is try with different repeat lengths and numbers of repeats, as well as different read lengths. And this is where you will start to see that as you increase read lengths, you start to resolve repeats with the OLC approach. You will find that as you increase read lengths, you will get to resolve these repeats, and they will not have these unresolvable knots or knot-like structures in the overlap graph. And this is where the long-range sequencing comes in.

So if you remember the first few classes, we talked about short read sequencing and long

read sequencing. So the problem with assembly that arises from the repeats can be resolved by long read sequencing and that is why we prefer to use long reads when you have to resolve these repeats. So I will encourage you to do this with a short reference sequence with different repeat lengths and different read lengths, and I will see that with increasing read length, you will be able to resolve these repeats better. So to discuss the drawbacks, we have mentioned two drawbacks of the OLC approach. So building the overlap graph through pairwise comparison is time-consuming.

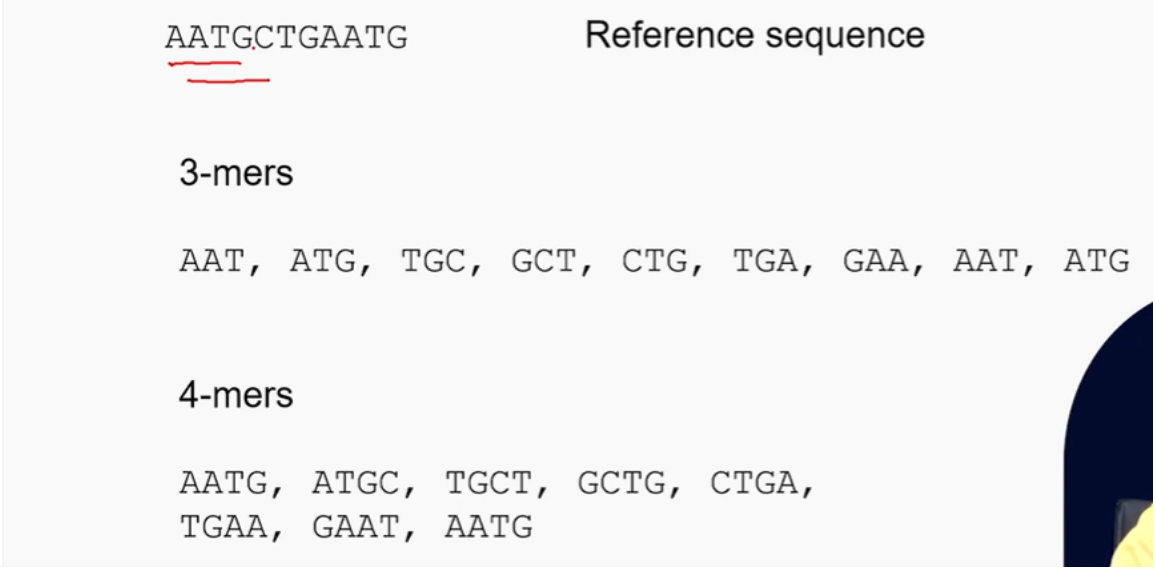
We are doing billions of these pairwise comparisons, and again, each comparison takes time whether you are applying dynamic programming or suffix tree-based methods. So that will take a lot of time. In fact, this is the most time-consuming step in the OLC approach. And also, the overlap graphs are quite big. So we have billions of reads; each read is present as a node in the graph.

So the structure is quite big, which would have to be stored in memory. So that will consume a lot of RAM. So with this in mind, we will now introduce this to de Bruijn graph assembly, or DBG assembly, and it is actually very different from all the methods that we have talked about so far. So the first point is that there is no overlap calculation from the pairwise comparison approach. So we do not need to do these billions of pairwise comparisons to calculate the overlaps or to identify the number of bases that are overlapping.

So this is not done with this method. Instead, what we do is build the graph that we build here using the k-mers from the reads. So this is an approach that we will discuss in much more detail. This is a k-mers-based approach where we get these k-mers calculated from the reads. So we will illustrate this with some examples.

So let us now illustrate this approach and how we built it. So here is a small reference sequence that we will start with, and we will calculate these k-mers. So what are k-mers? So these are substrings of length k. So if you take this reference sequence and want to calculate the three mers, you can start from here. So you can see we start from here and

take these three mers, and these are the three mers for this sequence.



Similarly, you can calculate four mers. So we start with this, and we take these four bases. So these are the substrings of the reference sequence of certain lengths, and this k can vary. Again, depending on the read length, you probably want to have the right k-mer size. So this is fine.

Now, how do we actually go about this assembly or the overlap graph? So how did you build this graph in the first place? So again, coming back to this reference sequence, let us consider this reach of length three. So we say k equals three, and the idea is very similar. You can try different Ks. So we will illustrate with k equal to three and k equal to four. So you understand this better.

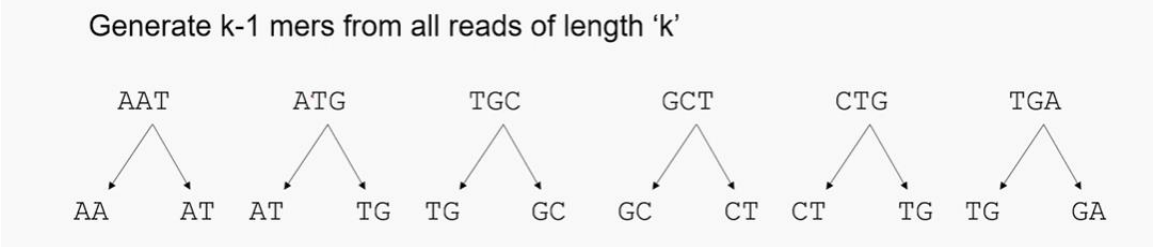
So for these k-mers, what we do is, in the first step, calculate or split these k-mers into two k minus one mer. So if k equals three, we have three mers. Then we need to calculate two of these two mers. So k minus one equals two. So we will calculate two mers.

So here is the splitting. That is how it works. So for these three mers, k equals three, we have the left k minus one mer, which is the first two bases, and then we have the right k minus one mer, which is the right, which is the 80. So we take the last two bases here. So it does not matter whether this k equals three or five.

We split the k-mers into k minus one mer. So we will illustrate this with k equal to 4. So you will understand this better. So once we have done this, we generate this k minus one mer from all reach of length k. So what we will do is take all these reach and split them into these k minus one mers.

So you have left k minus one mer and the right k minus one mer for each of them. So this is what we are doing in this step. So we start with 80. The left one is AA, and the right one is 80. For ATG, the left one is 80.

So the first two bases, as you can see, are the left k minus one mer, the right one is TG, and so on. So for TGC, the left one is TG, the right one is TC. So you understand how we are deriving this k minus one mer, and we do this for all the reads that we have in our data. So this is what we are doing. You can see all the reads we are splitting into left and right k minus one mer.



Now one of the things you might ask is, Why are you doing this? What is the point? Why are you splitting these reads into k minus one mer? So once you have split this, you will notice that these left and right k minus one mers share an overlap of length k minus two. So for this one, you can see this A base is common between these two, right? So here, k minus two equals one, right? So overlap is of length one. If k minus two is five, the overlap will be of length five. We will see with more examples in this class and the subsequent class that the overlap is k minus two. So now we know that overlap is k minus two.

We can now build this graph. So how do you build this graph? Because these two k minus one mers share this overlap, and for this k minus one mer, the suffix overlaps with the prefix of this one. And with that idea, we can build an overlap graph, and this is how it would look, right? So this is the left one, and this is the right one. The arrow direction

would be from the left to the right, right? Because of the nature of the overlap, right? So for the left, the suffix overlaps with the prefix of the right k minus one, okay? So this overlap is of length k minus two. In this case, it's one. The edge that we see, if we actually traverse through this edge, corresponds to this more AAT, okay? Because if we merge this, the overlap is one base.

So if we merge this, once we traverse, we can merge these nodes, and this will correspond to the k-mer or the read AAT, okay? So with this in mind, now we can actually apply this to all kids, okay? And we can generate this directed graph from all the reads, okay? So what you see here is no overlap comparison. What we just do is split the reads into k minus one merge, and from that, we build the overlap graph. And once we split, we know there is overlap, right? Because this left and right k minus one merge will share an overlap. We utilize that property to build that overlap graph, okay? So there is no pairwise comparison. We just split all these reads into this k minus one merge, okay? So this is what we have done, right? So these are the k minus one merges for all the reads we have split, and now we'll build the graph, okay? And this is a different graph, okay? So for the first one, we have the overlap.

So this is from this KMR or this read, right? We have this connection from A to AT. Now we move on to the next one, right? So this one is ATG, right? So AT, when you do this, right? For the first one, it's okay. There are no nodes in the graph, okay? So we define these nodes as AAT, and we add this connection. For the second one, right? We have these two k minus one mer. So first, we check whether this node actually exists in the overlap graph already or not, okay? And the answer is yes, right? We see this node already exists here in this overlap graph.

So we just add this connection and create a node for TG, right? So we also asked, right, whether this node TG exists in the overlap graph; it does not. So if it does not exist, then we create a new node and add the connection, okay? And we don't need to give the overlap because we know the overlap is the same for all of them, right? So this is k minus two for all the edges, right? So we don't need to add this weight or the edge weight in this overlap

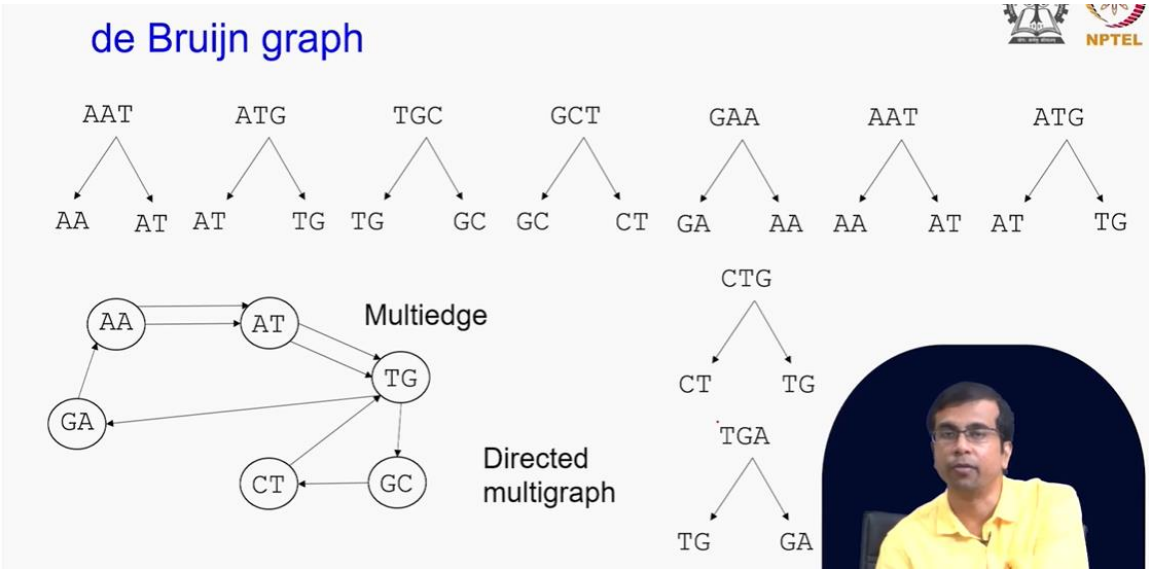
graph because they are the same for all the nodes, okay? So we move on, right? So the next step is the same, right? For TG, we ask whether there is a node, and it is there, right? And for GC, we also ask whether there is a node. If there is not, then we create a new node, right? And then add this connection, okay? So this is the connection that we have created for this one. In the next step, you can probably guess, right? So for GCT, this one, we again ask whether GC is there, GC is here, and then CT is not there in the graph. So we have to add a new node and add this connection, okay? So up to this point, we are done.

Now we are moving to the next graph, this k minus one mer, and we'll see, right, how we can add them. One of the things you might realize is that these are kind of present in order, so that's why they are appearing nicely one after another. But in the real world, they will not appear like this, right? So when you are considering reads, they will not appear like this, so they will appear at random, and then based on this, you have to define nodes and add connections, et cetera. But as you will see, this deep ruin graph construction is not dependent on the order of reads, right? So whether the reads appear at the beginning or the end, that will not influence the deep ruin graph. We'll see this in a moment, right? So now I change the order, okay, intentionally.

What you see is that now we have this G.A.A., okay? And we have this G A and then A A. So here we see A node; it actually exists, right, in the network. But the G-A node does not exist, okay? So what we have to do is define this G-A node and then add this direction, okay? So the direction is always from the left to the right, k minus one, right, for the reason that we have already discussed, okay? So what we do is add this, and we can now go into this one, A.A.T. Okay, so for A A T, what you see is that we have A A k minus one mer and A T k minus one mer. If you check the overlap graph, what you will see is that both of these nodes exist in the graph, right? You have A, and you also have T.

What do you do? So we don't need to define any nodes; we'll just add another edge, okay? So this is what we are going to do, okay? So we'll add another edge, okay? So you can have multiple edges between nodes, okay? So this is kind of getting to the repeat part, right? So if this part is repeated, you'll see these multiple edges. And similarly, we can go to the A T

G node, K mer, and then we see, okay, these two nodes again exist, so we add another edge here, and we'll have two edges between the A T and T G nodes, okay? Moving on, we can now come to this C-T-G one, right? And what we see is that these nodes actually exist: C, T and T G, of these K minus one mer actually exist as nodes in the graph, okay? So what we need to do is simply add this edge. Okay, add this connection. And finally, we have this one, T.G.A. And again, we can check whether these T G and G A exist in the graph; both of them do, right? So then we add this edge, okay? So this actually completes the overlap graph, okay? So this is kind of done, right? And one of the things we'll do is then we'll expand this, right, and say, do this for a different k-mer, and you'll get a better feeling for the whole process, okay? We'll talk about how we actually use these to generate the reference sequence a bit later, okay?



So one of the first things you notice is that in this directed graph, we have multi-edges, right? So this is called the directed multi-graph because you can have multiple edges between two nodes, okay? This is unlike the other ones that we have discussed so far. Okay, so let's take the same reference sequence, but now we define K as equal to four, okay? So we'll do this for each of length four, right? And the step is the same, right? So we follow the same steps.

The first thing is that we define this K minus one mer for all of them, okay? So we'd have to define this K minus one mer for all of these reads, okay? And as you can see, this K minus one mer is now of length three, okay? And then on top of that, right, you see the

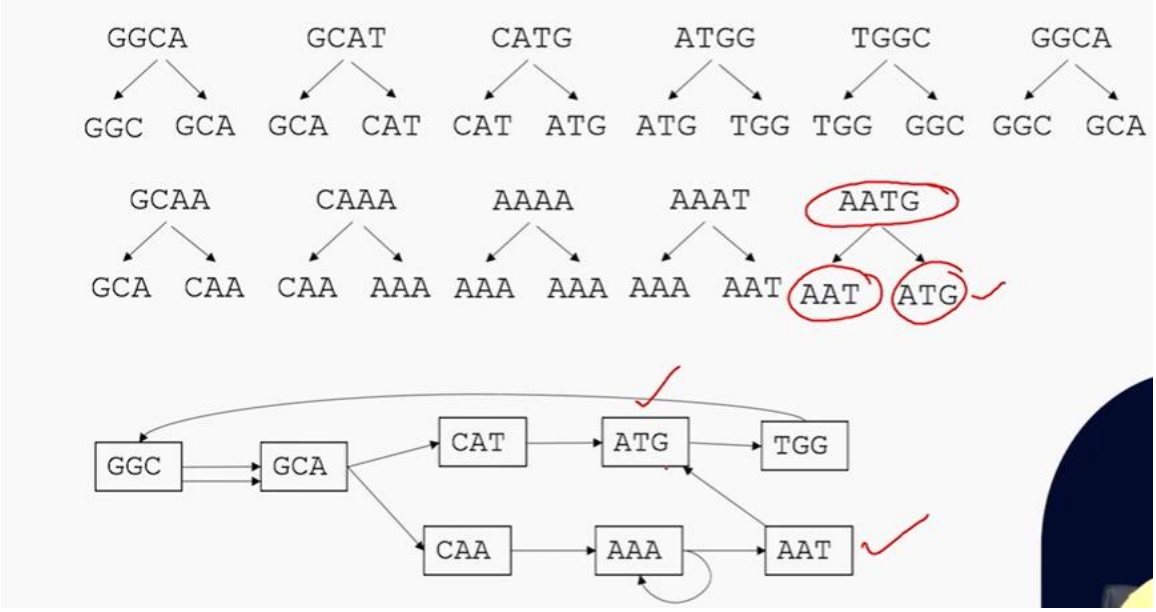
overlap is two bases, right? So K minus two is two. So your overlap between these K minus one mer is K minus two or two, okay? You can see this across all these K minus one mer, okay? Now we can take this K minus one mer, right? And we can now build this directed multigraph following the same process that we just described, okay? So let's start with this one, okay? And we have this GGC GCA, right? So this will be the connection, okay? Now we move on to the next one, right? The GCAT GC exists in the graph, and we then add the CAT node and this edge. Remember, traversing this edge will give us the K-mer, okay? The read sequence

So we proceed, right, with this. We have now moved on to the CATG, okay? And again, CAT exists, so we add this ATG node, and we also add the connection or the edge between these two nodes, okay? And so on, so we are here now, okay? We're just adding this TGG node. We add the edge. Now coming here, right, when you come to TGGC, we have these two K minus one mer, TGG and GGC. What we see is that both of these nodes actually exist in the graph already. So what we need to do is just add this edge here.

You can see this edge here, right? So from TGG to GGC, So we continue, right? We have to continue this process until we reach the end, right? Until we have covered all the K-mers in our data, okay? So from GGC to GCA, this actually exists. Both these nodes exist, right? And there is a connection already, so again, this will be multi-edge, right? So we need to add another edge here. So you'll see this one for GCA now, right? So for GCA, GCA exists, okay? And then we define the CAA node and add this connection. So now the graph is actually becoming interesting, right? So you have multi-edges; you have this bifurcation, right? There are two parts we can go to, okay? And then we come to this point here, okay? Where you have the left K minus one mer and the right K minus one mer, they're exactly the same.

The moment they're exactly the same, it means this is a self-edge, okay? So the edge comes out of that node and goes into that node, okay? So it's saying, Okay, this is present twice, right? So this K minus one mer is present twice, okay? So this edge again represents the overlap between these K minus one mer, okay? So if you kind of interpret this edge, this

will be the K mer coming out of this edge, okay? So now hopefully you understand, right? With this concept, we can have multi-edges and self-edges in this deep-read graph. So we next move to this AAAT sequence, and we see whether this AAAT, actually this K minus one mer, exists in the graph, right? And we see, okay, AAAT, K minus one mer already exist as a node, and we need to add this AAAT, and we add that at the edge of this graph. And finally, we have moved to this AATG. Again, we do the checks for this K minus one mer.



So, whether it exists in the graph, it does. And for AATG as well, it does exist in the graph. So we add this connection to this edge in this graph, okay? So now the directed multi-graph is complete because we have utilized all the reads in our data. So the question is now: how can we use this directed multi-graph to actually determine the reference sequence? So this is the idea, right? This is great; we have the directed multi-graph. Now how we can utilize this to generate the reference sequence, okay? So the question can then be translated into something else, right? So if you remember, each edge actually represents a K-mer, right? So can we find a path in that graph that traverses through each edge exactly once? If we can find a path like that, that will actually give us the reference genome sequence, right? So because each K minus one mer is, each K-mer is represented by the edges, right? So if you can traverse through each edge exactly once, we have utilized all the reads, and we can generate the reference sequence, okay? So with that idea, we can look at some of the properties of the graph and see whether we can observe any such path in the graph, okay?

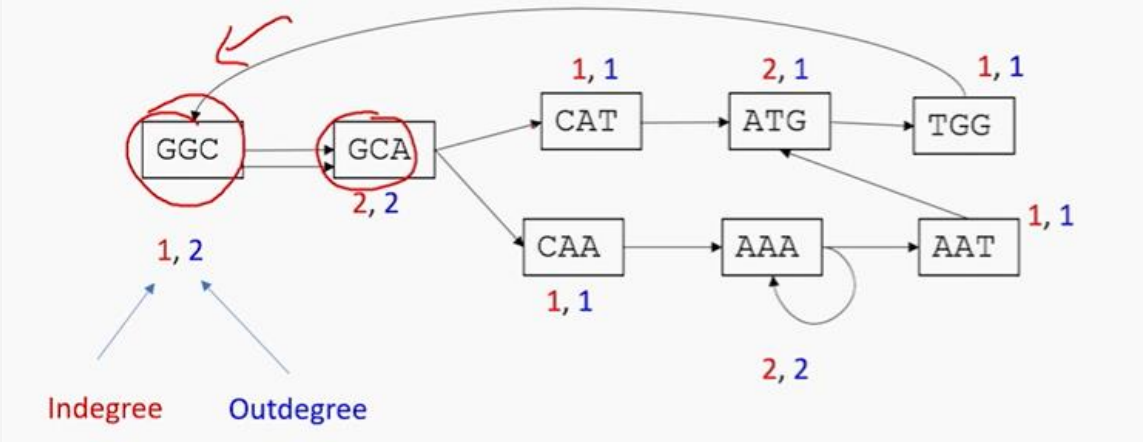
So before we go there, we will define certain terms.

The first term is the degree of a node. So the degree of a node in a graph is defined by the total number of incoming and outgoing edges. So if you take this small example here, right? So you have this node GGC and another node GCA, so the degree is two for both of them, right? So for GGC, what you see is that there are two connections going out but no connections coming in, right? So it's incoming plus outgoing, so the degree is two. And similarly, for GCA, you have two incoming connections, right, but no outgoing edges, so the degree is again two. Now we can also define something called the in-degree of a node. So this is the number of incoming edges, or the number of incoming connections.

Also, we can define the out-degree of a node, okay? So these are the numbers of outgoing edges. So the degree, as we have discussed, is just simply in-degree plus out-degree, okay? So now we can define for this very small example the in-degrees and out-degrees. As you can see, for the node on the left, the in-degree is zero because there are no incoming edges, and the out-degree is two because there are two outgoing edges. And for that node on the right, the in-degree is two because you have two incoming edges, and the out-degree is zero because there are no outgoing edges. Okay, so now moving on to the full graph, right, the full overlap graph or the directed multigraph that we constructed, we can now define these in-degrees and out-degrees for every node in the graph, okay? So with that in mind, right, you can now color code these in-degree and out-degrees, right? So we can say red one or the first number is in-degree and the second number is out-degree, okay? So for this one, as you can see, you have just one incoming edge, right? So this one is here, so the in-degree is one, and you have two outgoing edges, so out-degrees are two. You can actually look for incoming edges and outgoing edges for each node in the graph.

So for this one GCA, it's two-two because you have two incoming edges and two outgoing edges. So similarly, you can define this for every node in the graph, right? For these ones, right, this is simple; they have just one incoming edge and one outgoing edge. Here you can probably see, right? You have two incoming edges and two outgoing edges, right? So for the self-edge, you have one incoming edge and one outgoing edge, right? So that's why

it's two, okay, in addition to the one that is coming in and going out from this node AAA, okay? And finally, for this one here, you can see two incoming connections, okay? So you have these two connections, right? One from here, another from here, so in-degree is two and out-degree is one, okay?



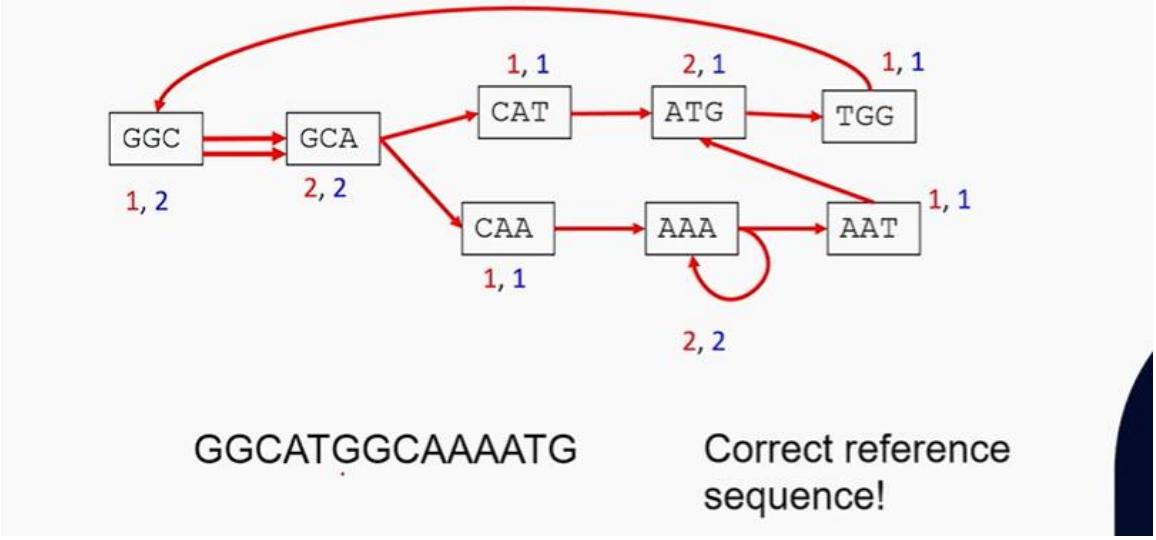
So we have now defined the in-degrees and out-degrees for each of these nodes, okay? So why is it important? What are you going to do with these in-degrees and out-degrees? So it turns out that we can define certain things that will tell us whether there is a path that we just want, okay? So for that, what we do is, with these in-degrees and out-degrees, if a node has equal values of in-degree and out-degree, that node is called balanced, okay? So we'll call a node balanced if in-degree is out-degree. We have seen certain nodes in our graph, right, where the in-degree is equal to the out-degree. In addition, a node is called semi-balanced if the absolute difference between in-degree and out-degree is one, okay? So in-degree could be higher than out-degree, or out-degree could be higher than in-degree.

That does not matter as long as the absolute difference in the value is equal to one, okay? So they can differ at most by the value of one, okay? So with this in mind, we can now check our graph and see whether we have balanced and semi-balanced nodes. Why are these important? Why are we interested in this balanced and semi-balanced node? So we'll actually introduce this concept of something called the Eulerian path, okay? So there is a path, right, in a graph. If that path exists, right, so this will be a Eulerian path. So this is a path that visits each edge of a graph exactly once, okay? This is something we wanted, right? This is the question we framed a few minutes ago, right? So this is the path we want, okay? This is a Eulerian path that will visit each edge of a graph exactly once, okay? And

there is a condition that has been seen, okay, that a graph is Eulerian, meaning there would be an Eulerian path if and only if it has a maximum of two semi-balanced nodes and the rest of the nodes are balanced, okay? So we find a directed connected graph where you have at most two semi-balanced nodes, okay, and the rest of the nodes are balanced. So we have defined these balanced and semi-balanced nodes just now, okay? So following that idea, if you have at most two semi-balanced nodes, we can say, okay, there is a Eulerian path in that graph.

So this actually helps us ahead of time, right? Before we try to find a path, the Eulerian path, we know whether or not such a path exists, right? So with this in mind, let's actually move to our graph, right, and see whether we can find an Eulerian path, right, whether this exists, whether this condition is satisfied, so which will tell us whether such a path will exist, and then try to find the Eulerian path, right? Because if we can find the Eulerian path, we'll get to derive the reference genome sequence, okay? So again, back to this graph, right? We have this in degrees and out degrees. What you notice is that most of these nodes show equal in degree and out degree values, right? So if you'd consider this one, this one, this one, right, these are all showing the same degree and our degree value, so these are balanced nodes, okay, as we have just defined. But for these ones, right, this one and these two, right, so these two nodes are not balanced, but as you can see, they are semi-balanced, right? So the difference between in degree and out degree is just one, okay? So these two are semi-balanced nodes, okay? So if you have these semi-balanced nodes and they are at most two in the graph, this means that there is a Eulerian path in this graph, okay? If you remember the condition, there could be at least, at most, two semi-balanced nodes, and the rest of the nodes should be balanced, so this condition is satisfied for this graph, which means there is an Eulerian path, okay? So now that we know there is an Eulerian path, we can actually search for it and traverse through that path, and that will give us the reference genome sequence, okay? So let's find the Eulerian path in that graph, okay? So usually, we start from a semi-balanced node where our degree is greater than our degree because that's where we'll have an extra out degree or an extra outgoing edge, right? So in this case, it's this node here; this is semi-balanced, and this out degree is greater than the in degree, right? So we start with this here, and as we traverse this edge, we are getting the reference

sequence, remember, right?



Each edge is a K-mer, right? So that's the sequence, okay? Now we have traversed through this, and now we have two options, right? We can take this direction or we can go in this direction. Okay, as you will probably see, right, if we come in this direction, we might not actually get the Eulerian path, but we can try again. But let's now traverse through one direction and see whether we get the reference sequence or not, okay? So here is this one, right? So we are going to this CAT. So we are also adding this to the reference sequence, and then we go to this ATG node, right? And we are adding this to the reference sequence here again, right, keeping in mind the overlap between these two adjacent nodes or the edge that we are traversing, okay? And as we traverse through these edges, we actually build up the reference sequence, okay? So I'm highlighting this traversal through these red arrows, as you can see, and we continue this process, right? And we are traversing every edge exactly once, right? This is very important here, okay? We need to remember this, okay? And here we traverse through the self-edge, okay? And then finally we have traversed through all the edges, okay, and this is the reference sequence that we have got after this edge, okay? And this actually turns out to be the correct reference sequence. But one thing I'll tell you is that there might be multiple Eulerian paths in a graph, and they might not always give you the right reference sequence, okay? So there are options, again, to find these Eulerian paths and to see which one would be the best reference sequence, okay? So here are the references for this class.

So we have discussed the DBG method and introduced how it works, right? So we see the reads of length k are first split into left and right k minus one merge, and these left and right k minus one merge have overlap of length k minus two. And we also generated this directed multigraph from these k minus one merges, which are then used, right, to actually determine whether there is an Eulerian path in that graph, and if that path exists, we traverse on that Eulerian path, which gives us the reference sequence. Thank you very much.