Next Generation Sequencing Technologies: Data Analysis and Applications

Assembly and Quality Control

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. We have discussed different algorithms for de novo genome assembly, starting with next-generation sequencing datasets. So, in the last class, I introduced the last algorithm, which was the de Bruijn graph assembly, and we talked about the steps that we needed to perform to get to the reference sequence. So, what we will do today is look into special cases again with this de Bruijn graph assembly, and then we will talk about the subsequent steps after we have done the assembly. So, this will be the agenda for today's class.

So, we will talk about DBG assembly with non-uniform coverage and sequencing errors, how these non-uniform coverage and sequencing errors actually affect the assembly process, and what the outcome is. We will then talk about the different tools that are available for assembly. So, these tools are based on the algorithms that we talked about. So, I will just briefly mention those tools that you can use, and then we will talk about the assembly quality.

So, once we have assembled the genome, we need to evaluate the quality of the assembly process. So, whether the assembly is actually good or whether there is some scope for improvement, those things can be assessed using this assembly quality process. So, I will discuss that, and finally, we will talk about something called genome annotation. Once you have assembled the genome, you want to identify the different features that are within it. For example, you want to identify the genes for it, or the codon coding sequences, exons, introns, etcetera.

So, these will be the keywords for this class assembly quality and annotation. So, let us begin with the de Bruijn graph assembly. I will simply highlight the steps. So, the first step is to start with the reads that are of length k, and we generate the left k minus 1 mer and the right k minus 1 mer for every read that we have in the data. The second step is to generate this directed multigraph based on this left and right k minus 1 mer because these left and right k minus 1 mer share an overlap of length k minus 2.

We utilize that property to generate the directed multigraph. In the third step, we actually look at the properties of the graph that we have created, and we talked about these balanced and semi-balanced nodes. Based on the fulfillment of certain criteria, we can check whether there is an Eulerian path that is available in the graph. So, this is something that will establish that there is an Eulerian path. So, we also talked about this Eulerian path, right? So, this is a path that traverses every edge exactly once, and as we traverse through the graph using this Eulerian path, we actually generate the reference sequence.

So, we derive the reference sequence by traversing through this Eulerian path. So, what will be illustrated today is that we will see that different graph assemblies may not always keep the right reference sequence, and we will start with the same example that we talked about in the last class. So, why would this happen? Why would you not get the right reference sequence? So, because in a graph that satisfies all the criteria, there might be multiple paths, It is not necessary that you have just one path that is unique, but you can have multiple Eulerian paths, and we will see in an example that these multiple paths can give rise to different reference sequences. So, again, these are highlighting the limitations of the assembly algorithms, and it is good to know this because then we can think about ways to actually address these issues.

So, let us start with the same example that we actually worked with in the last class. So, here is the reference sequence on top. As you can see, the reference sequence is the same reference sequence that we talked about in the last class. Then we have reads of length 4, okay? So, here the k equals 4 right, and I have written down all these 4 marks for this reference sequence. So, what we do now is, from these k marks, generate the left and right k minus 1 marks.

So, as we discussed, this is the first step. So, that is what we do, and from that k minus 1 mark, we generate this directed multigraph, ok? We generated this directed multigraph in the last class. So, I am not going to go into those steps again, ok? So, what I am showing is that this is a directed multigraph with nodes.

So, these nodes are k minus 1 mers in the connections; these are the edges right, showing the overlaps, and they actually signify the k mers. So, as we traverse through an edge, we get a k mer. In addition, I am showing these two numbers, which we also introduced in the last class. So, the first number is the number of incoming edges, right? So, this is the in-degree for a node, and the second number in blue is the number of outgoing edges of the out-degree for every node.

So, we are using this convention to denote this in degree and out-degree for this node. Now, why are we considering this in-degree and out-degree? Because these are important to decide or derive whether there is an Eulerian path available in this graph. And the condition was that there could be at most two semi-balanced nodes, and the rest of the nodes should be balanced, again reminding you about balanced and semi-balanced nodes. So, balanced nodes are the nodes where the in degree equals the out-degree. So, in this graph, you see all these nodes here right; these are balanced nodes, ok?

So, the in degree and the out degree are identical, but then you have two other nodes, ok, this one here and this one here, where the in degree and the out degree are not the same, but then the absolute difference between the in-degree and the out-degree is equal to 1. So, this means these two nodes are semi-balanced, okay? So, this graph then satisfies the criteria that we define as right. So, you have at most two semi-balanced nodes. So, we have two here, and the rest of the nodes are balanced.

So, this means there must be an Eulerian path in this graph, ok? So, this is the right condition, which is satisfied. Now, we have to find the Eulerian path, okay? And last time we traversed through this graph to actually find the reference sequence, I just mentioned that we could have an option, but we took one path, and we actually got the right reference sequence. So, if you go back to the last class, you will see that this is the path we took right, which I will highlight with these red arrows. As you can see, we are traversing between these two nodes, GGC and GCA, and we got this GGC right.

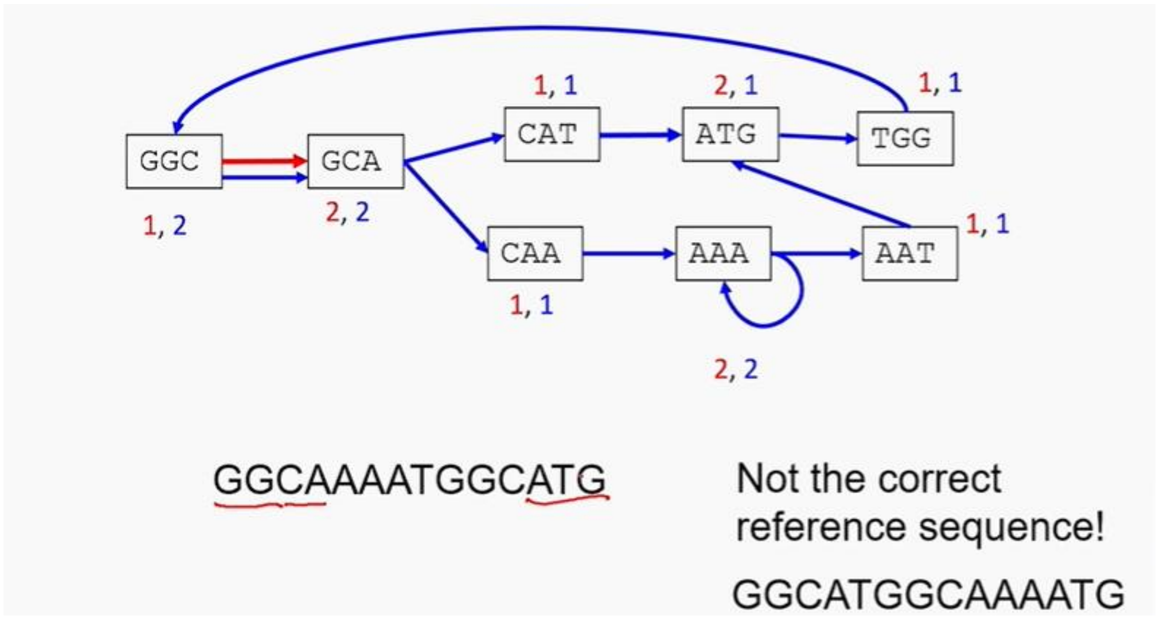
So, this is the reference sequence that we are building as we traverse through this path, right? So, every edge actually gives back the KMR. So, we took this path last time, right? So, we are going in this direction from GCA to CAT with this red arrow, then from CAT to ATG right. And as we go along, we are also building the reference sequence. At the bottom, you can see that I am just adding the letters.

And then we completed this process again without going into the full details. We completed this process right, and we actually traversed through the full graph. Finally, we reached here, and that is the end, and we got back the right reference sequence. So, this is something we explored in the last class. Now, the question is: are there alternate paths in the graph? We start at the same point, right with that red arrow. So, if we start at the same point, is there any other path that is also Eulerian? So, let us look at these options, right?

So, here you have these two options. So, last time we took this option right here we took this branch, but now we also have this branch. So, what happens if we traverse through this branch, which is highlighted by this blue color? So, a blue-colored arrow you can see. So, if we traverse through that branch, what happens?

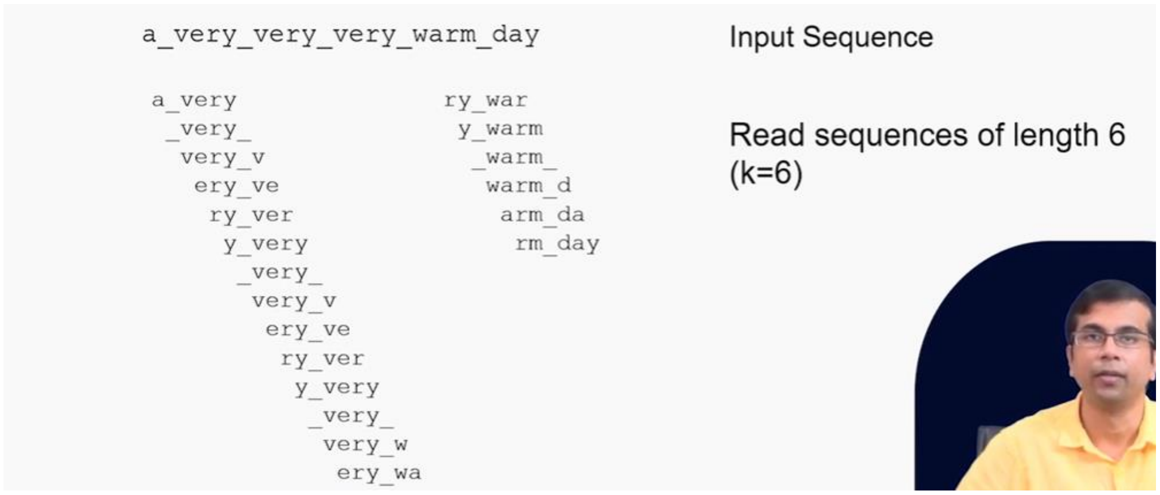
So, let us traverse through that and let us see whether there is an Eulerian path, and if there is, then let us find the reference sequence for that path. So, we traverse through this now I will show you step by step again now highlighting with blue arrows ok and this is the path that we are taking we are also building the reference sequence and then we take the self-edge right we are adding these letters to the reference sequence then we take the next step then the next one right we are going through this right this these are the solutions from this ATG now we have to take this to TGG right that is the way the arrows are given right in this graph So, we go to TGG, and we then go back to this GGC, right? So, that is where we started, right? So, we have now again this option, GGC to GCA, where there are two edges.

So, we are taking the second edge, and now we are traversing through the second edge. In essence, we are kind of now utilizing all the paths that are available or traversing through all the paths that are available. Now, from this point, we can take the upper path. So, towards CAT right because we have already taken the downward path. So, we take the upper path CAT, and then we reach the final node ATG. So, we have traversed every edge exactly once in this graph, which means this is an Eulerian path, and corresponding to that path, we have the reference sequence.



Now, we can compare this to the reference sequence that we started with, and it turns out that this is not the correct reference sequence. I have written down the correct reference sequence that we got earlier, and you can see this difference right there. So, the first two first three are actually identical, and maybe the first four are identical, right, and then the last three are also identical, but then in between there are some changes, ok, and you can see the order is different in this AAATGGC. This order is different between these two reference sequences that we derived from the same graph, ok, and that is because of the specific structures maybe some elements are repeated. So, that is where we are getting this kind of structure where you can have multiple Eulerian paths and we end up with different reference sequences ok? So, this is something that we should keep in mind. It is not always necessary that we have a unique reference sequence derived from a multigraph. So, now what we will do is take another example where we will actually do the assembly with repeat sequences, and we will see how the Bruijn graph actually handles them.

So, we will take this input sequence. We have used this input sequence before for the OLC approach, etcetera. So, we will take this example of a very, very warm day. So, we see this repeat very clearly; you have this very, very part right. So, this word is repeated three times in this sequence. So, this is the repeat region, right? Let us build the de Bruijn graph, ok?



```
a_very_very_very_warm_day

a_very              ry_war
 _very_              y_warm
  very_v              _warm_
   ery_ve              warm_d
    ry_ver              arm_da
     y_very              rm_day
      _very_
       very_v
        ery_ve
         ry_ver
          y_very
           _very_
            very_w
             ery_wa
```

Input Sequence

Read sequences of length 6 (k=6)

So, we work with a sequence of length 6 as an example; you can try with other lengths as well; that is not an issue. So, the process is exactly the same, ok? So, here are the reads that you can find in this sequence, and you will probably notice some of the reads are repeated. So, because they are coming from the repeat regions, and one of the assumptions that we are making is that every region is represented equally, What it means, in other words, is that we are expecting uniform coverage.

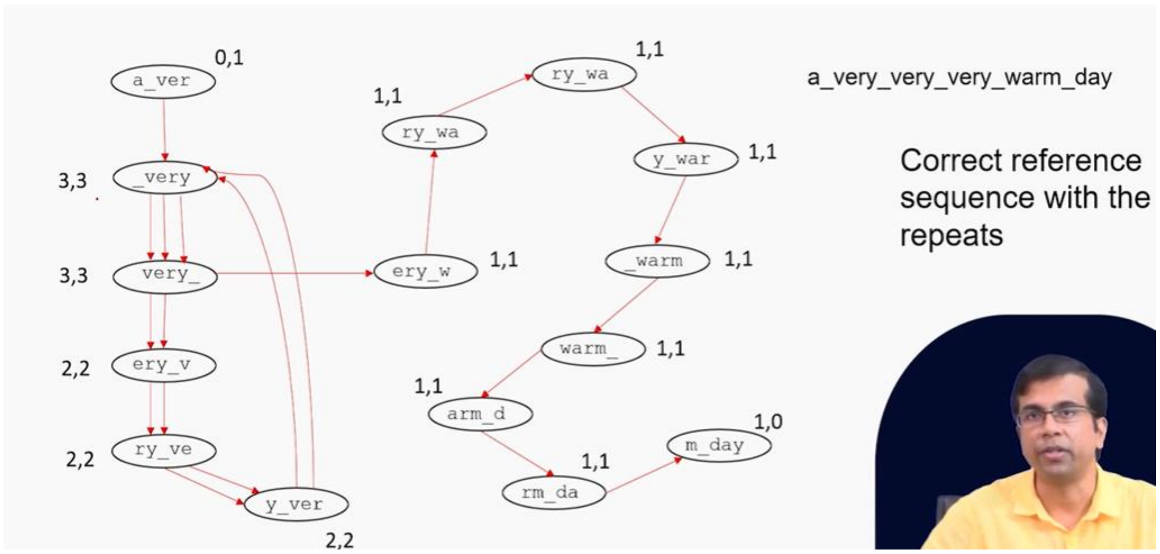
So, every region of the genome is covered uniformly. So, we expect uniform coverage, and we will see later on what happens if we deviate from that assumption. So, with this ideal scenario, we now build all these left and right k minus 1 mer ok. Of course, I would not have time to describe all this. So, you can see them on your screen. You can take a pen and paper, and then you can derive all these left and right k minus 1 mer yourself by following the same process that we described before.

So, once we have built this left and right k minus 1 mer, we can now build the directed multigraph, and this is how it looks. So, again, the process is exactly the same as before what we have discussed, ok? We add these edges between left k minus 1 mer and right k minus 1 mer, and we kind of keep on adding edges right, and we end up with these multi-edges, and until we exhaust all the reads that we have, ok. So, if we follow the same steps now, we can identify the balanced nodes or semi-balanced nodes by pointing out the in-degree and out-degree values for every node in the graph. And as you can probably see in this figure, you have two semi-balanced nodes.

So, here is one right This is semi-balanced, and here is another one, ok? So, these two are semi-balanced because the in-degree and out-degree absolute differences are 1. For the rest of them, they are all balanced because a degree is equal to our degree, ok? So, we have two semi-balanced nodes, which means there is an Eulerian path in this graph.

And what we have to do is then traverse. Through this graph, we can trace this Eulerian path and derive the reference sequence. So, let us do that right. So, we start from this point right again. We start with an edge that is semi-balanced and where the out-degree is greater than the degree right. So, we start from here, okay, and we are traversing again. I am highlighting these paths through these red arrows, and we will kind of show this very simple step because this is the kind of straightforward path right. So, we traversed up to this point; now we go up, right?

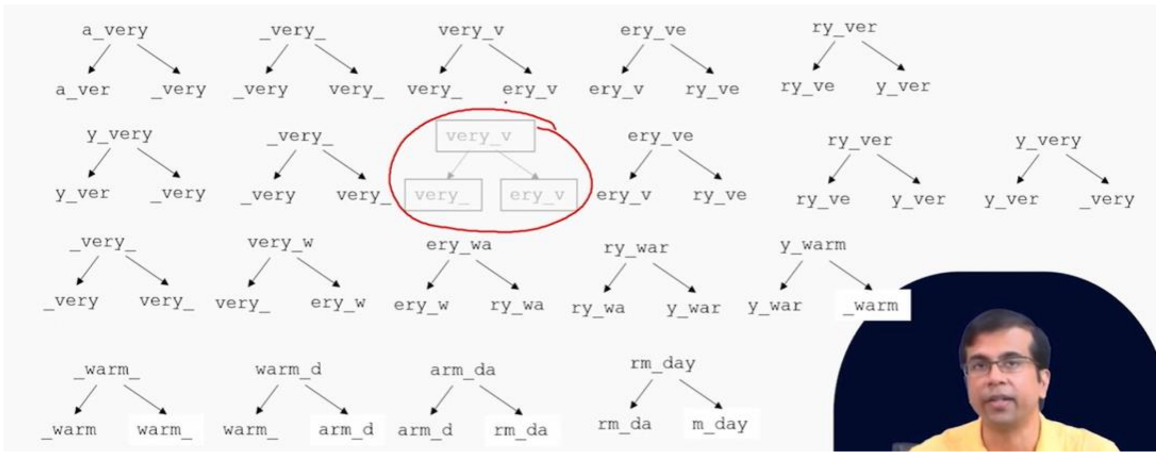
So, we are taking this path here; we are going up here up to this point again, ok? And then we traverse to another edge, right? So, another round of reversal through this region right through these nodes and then we go up again right using the black ones. So, black ones we have not traversed yet, and then we come back, come down here to this node right, and now we have to take the right.



We have completed the traversal of this part. So, we can take the right now, and once we take the right, you can see this is very straightforward. We have this simple path that can traverse through all these nodes, and we end up with the reference sequence on the right. So, as we are traversing, you will see I am writing this reference sequence on the right, and we are getting the reference sequences derived as we proceed through this graph. So, what do you notice in this case? So, we have derived the reference sequence, including the repeat sequences. So, this is the correct reference sequence with the repeats that are available.

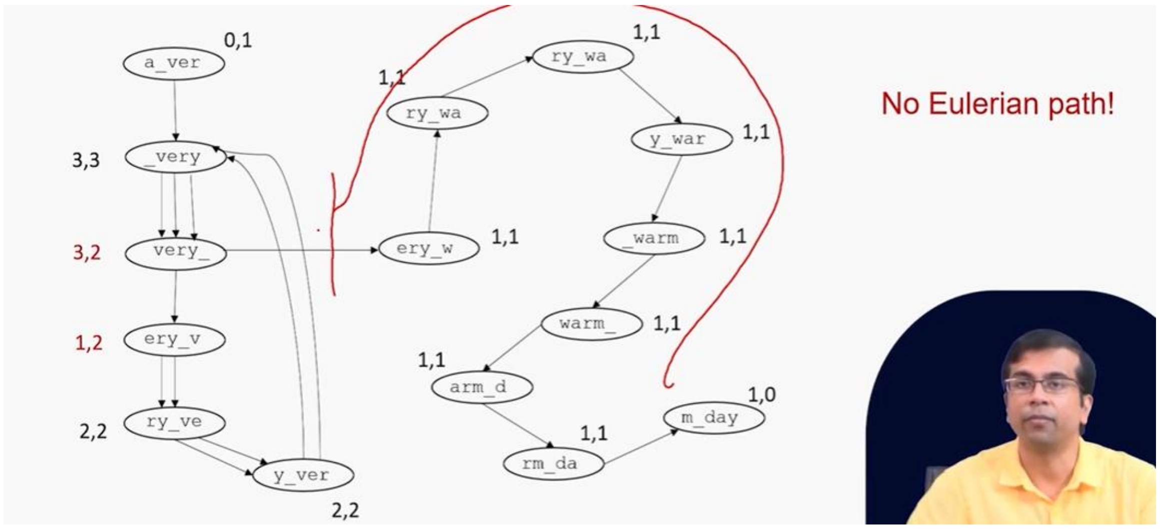
So, what you can see, at least from this ideal example, is that the right DBG assembly can resolve repeats to some extent, but then the assumption is that all regions are covered uniformly by the reads. So, this is a very critical assumption, right? The moment we shift from that, the whole thing will change. Now, why am I saying this is critical? Because in real life, it is very unlikely that you will get uniform coverage from all the regions in the genome, and in that case, you might not be able to find Eulerian paths on the graph. So, what will happen if we have non-uniform coverage? We will see that this can lead to graphs that do not have any Eulerian path.

So, this is the example we will take. We will do the assembly with non-uniform coverage, ok, and we will take the same example. What I will do is, I have also turned down the read sequences again, as before, of length 6, to generate this non-uniform coverage. What I will do is, like, remove one read from the data. So, which I have highlighted here, you can see right this read is removed. I have removed this read right. So, this means this will create this non-uniform coverage because a part of this reference sequence will not be covered by the reads. So, what happens now? So, we go through the same process I have.



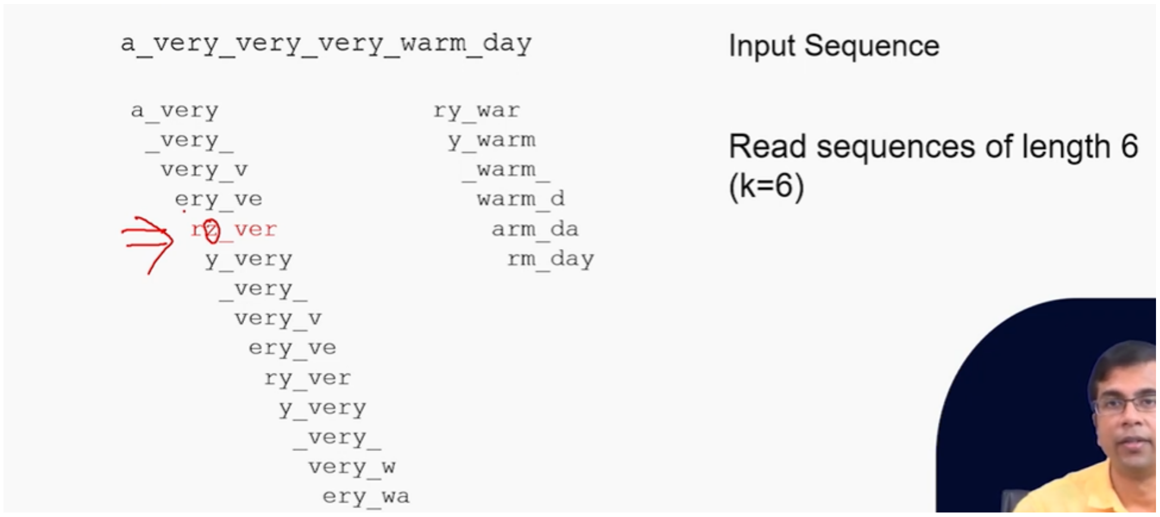
So, remove this k minus 1 mer construction for this read because this read is not present anymore, and we can use the rest of the k minus 1 mers to create the multigraph that we created as before, ok? And once you create this right, what you will notice is that from this one right connection, this one of the connections will be removed right because this read is not there anymore. So, these two k minus 1 mers will not be there in the graph, which means we will have one less edge here between these two nodes. Now, the moment you have this difference removed, what you see is that we have created two other nodes that are also not balanced. So, we have had two nodes that were semi-balanced right now, we are not balanced, but we have now created two extra nodes that are not balanced either, ok, because of this removal of this edge.

Now, you can see the in-degree is not equal to the out-degree; right, they are semi-balanced; the difference is 1; the absolute difference is 1; ok. So, this means that in this connected graph we have four semi-balanced nodes, ok, and this means we cannot say that there is an Eulerian path, right? So, there is no Eulerian path because we are not satisfying the criteria that we discussed before. So, for an Eulerian path to exist, the graph should have at most two semi-balanced nodes, but here we see four semi-balanced nodes.



So, this is the problem now. So, if you do not have an Eulerian path how do you define it and derive the reference sequence? So, that is the problem that we end up with, and you can probably see now that we will not be able to derive the reference sequence. Maybe some part of the graph will be Eulerian; maybe you can have this; you can disconnect certain parts; you can probably see right; maybe this right part is here; if you remove this part right from this edge and you can see this part, this part of the graph will be Eulerian. So, you can derive the reference sequence OK, and similarly, you can see whether certain parts of this graph are Eulerian, and maybe you end up assembling fragments of the reference genome OK. So, these fragments are the contigs that we discussed before, right?

So, these are what we usually get at the end of the sequencing process right at the end of the assembly because of these reasons, but because you will not get equal coverage or uniform coverage across all regions in real sequencing experiments, ok. So, we will talk about another scenario. So, we have to do assembly with sequencing errors, right? So, this is quite common because the sequences that we work with make errors. If you are working with short reads, there will be errors; if you are working with long read data, etcetera, you will also have errors and a lot more errors. So, this is something that we would have to deal with in the real part, where we cannot just apply that de Brujin graph assembly in the ideal word.



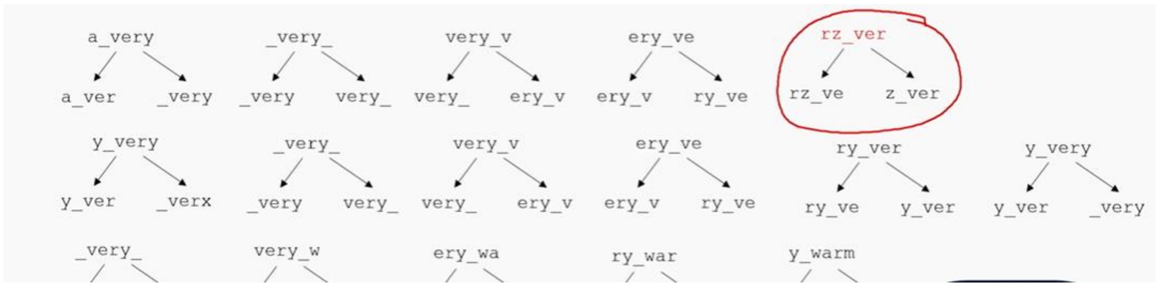
```
a_very_very_very_warm_day

a_very              ry_war
 _very_              y_warm
  very_v              _warm_
   ery_ve              warm_d
    rz_ver              arm_da
     y_very              rm_day
      _very_
       very_v
        ery_ve
         ry_ver
          y_very
           _very_
            very_w
             ery_wa
```

Input Sequence

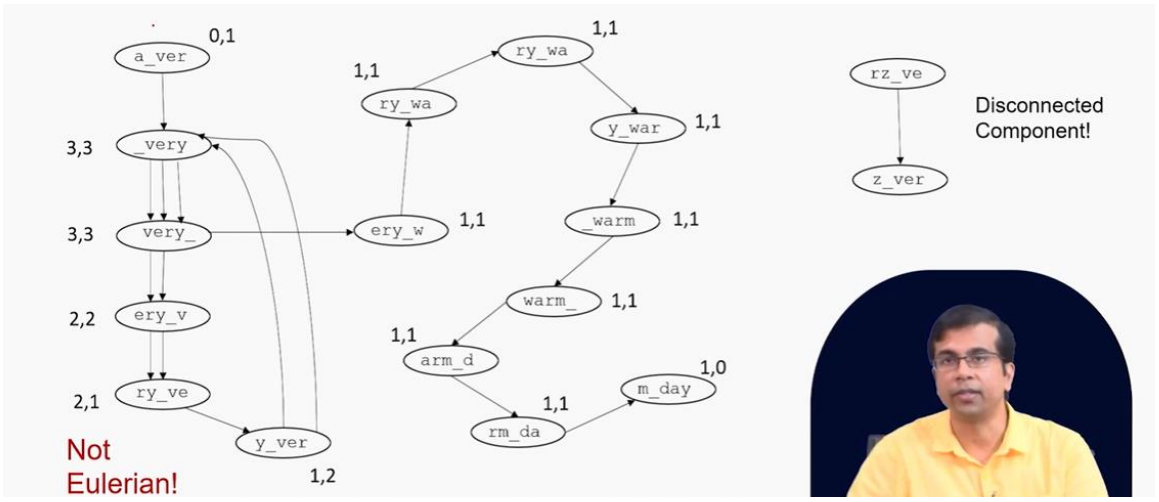
Read sequences of length 6 (k=6)

So, we have to deal with these issues in the real world. So, let us see what happens if we have a sequencing error in one of the reads. So, for doing that, we have taken the same example, ok, with the same reads as what I have done here for this one read here. I have changed the letter instead of this, and I have now made it rz ok. So, this is a sequencing error; this z did not exist in the reference sequence, and we have a different letter here. So, let us see how this will impact the deBrujin graph assembly process. So, again, as before, we are getting the same k minus 1 mer. Now here is the read with the mutation right; we have this rz underscore ver right, and we have derived the k minus 1 mer left and the right ones, ok.



So, what we are going to do with this is now construct the directed multigraph. When you construct the directed multigraph, what you will notice is that this rzz underscore ve does not exist, and this k minus 1 mer does not exist in the graph. So, what will happen is that these two k minus 1 mer will form a separate graph, which is disconnected from the main graph, and on top of that, what you will notice is that we have again created two nodes that are not balanced. So, we already had two earlier nodes, one at the beginning and one at the end, but in addition, we have created two other nodes that are not balanced. So, again, you cannot find an Eulerian path in this directed multigraph, ok?

As before, you can probably disconnect certain parts and see if that part of the graph is Eulerian, and perhaps you can derive the reference sequence, and on top of that, you have this disconnected component. So, this is something we cannot do anything about, and we might then have to simply keep this kind of component; we cannot really derive anything.



So, as you can see, these sequencing errors can again disrupt this assembly process, and as I mentioned in this part, this biggest component is not Eulerian because of the reasons that we just discussed. So, one of the other points that I just wanted to mention is that in deep brain graphs, we also sometimes use edge weights. So, this is something that we do because if we have, let's say, an average coverage of 30, 50, or 100 X for a genome, you do not want to have these 100 edges track right.

So, instead of drawing all these edges or storing this information, we can simply put a number saying there are 100 edges. So, those connections, the number of edges, the arrows, or the number of edges could be the weights of those edges, right? So, I will just illustrate, right? So, this is actually done for coverage, where you have this high coverage for genomic features. As I mentioned, it can be 50 to 100 X, and instead of drawing or keeping track of all edges, when you are doing the computation, you can give one edge weight for each distinct k-mer.

So, this k-mer is the edge, right? So, the edge represents a k-mer that we have discussed.

So, I will just illustrate with this example. So, this is the same example as before: we have these edges now. Here the numbers are small, but you can imagine if instead of these 3 or 4 or 2 right, you have let us say 30. So, we have to keep track of all this information; it is actually better than to actually convert them to numbers and say we have 3 here; this will automatically signify we have 3 edges here, ok?

So, this is something that can be done, ok? So, to summarize what we have discussed about de Brujin graph assembly, So, DPG assembly does not require pair-wise compression of reads, which actually saves time, and this means this algorithm will be faster than the other methods that we discussed, namely OLC or SCS. So, what we have also seen is that if a graph satisfies these criteria for Eulerian paths, we can get the reference sequence by traversing through the multigraph, and we have also seen that there are different scenarios where you can have multiple Eulerian paths and might end up with the wrong reference sequence. And finally, we have also seen that if you have non-uniform coverage repeats and sequencing errors, this can affect DBG assembly. So, we took some illustrations, and we have shown that this can disrupt the assembly process, and of course, there are ways we can address this one. Of course, look for subgraphs that can be Eulerian or maybe correct for sequencing errors.

So, maybe if you are working with long read sequencing data as well as short read sequencing data you can use the short-read sequencing data to correct the long read data. So, that will reduce the error. So, you can take different approaches to address these issues, ok? So, what do we get after we have done this denovo assembly? So, whether we have used the OLC approach, the DBG approach, or the SCS approach, whatever the approach, what do we get at the end?

So, we usually get multiple contigs. As I mentioned, right repeats remain in many cases, and even with DBG assembly, these repeats will persist because of non-uniform coverage. So, we end up with these multiple contigs, and there are gaps between these contigs, ok? And to cover this, we do something called scaffolding or gap closure. So, this is usually done between paired ends using paired ends, right?

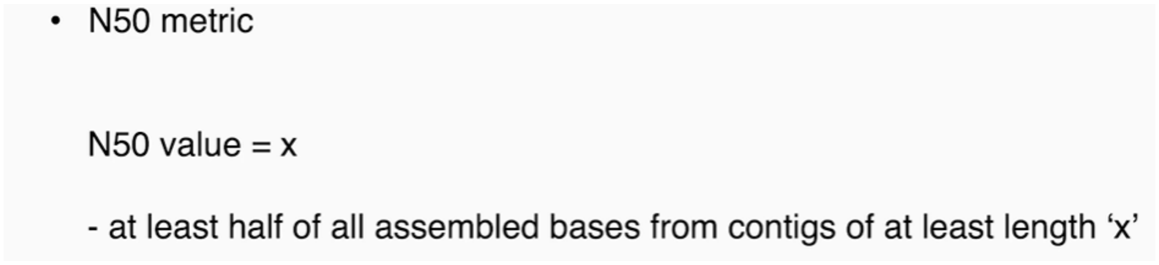
So, imagine you have these two contigs, right? So, this is contig 1, and this is contig 2, right? And in case you have some reads between these contigs, So, with this paired end data, you can use those to actually do the gap closure, or what we call scaffolding. So, as you go around and go about this assembly process, you will see you can have these different levels of scaffolding. You can specify what level of scaffolding you want, etcetera. And here is a reference describing some scaffolding algorithms that can be used for this purpose.

So, I will just now mention some of the tools that you can use for the assembly. So, we have discussed the algorithms, and these are implemented in different tools. Of course, they are based on some of these algorithms, but they take slightly different approaches and some different implementations. And the early tools for assembly were the Celera; this is the assembler used for human genome assembly again based on the OLC approach. And then you also had the Newbler, which was the 454 sequencing assembler.

For assembly today, you have velvet available. So, this is based on the print graph method that we discussed. Here is a link and here is the reference paper that you can go through. Similarly, you have other tools; I am just mentioning some of them, which I will not discuss because we have discussed algorithms and the implementations might be slightly different. I will just give you references in case you want to read more and understand these tools better. You will find the links here, ok? So, here is another tool called Hinge; this is an OLC-based approach, and here is a link and the reference paper. And similarly, you now have other sets of tools that are based on De Bruijn graph assembly.

So, one is SOAPdenovo version 2, then you have SPAdes, again, this is the De Bruijn graph-based assembly method. We have another one, which is the IDBA. It actually has an interesting extension that can work with highly uneven coverage or depth, as you can see in the title of the paper. So, this is again a very interesting extension of the original tool, which is the IDBA. And finally, there are also other tools that can actually utilize other types of methods; we have not discussed other types of algorithms or extensions of those algorithms. So, here is one, which is called the SGA. This is a string graph-based assembly, and here is the paper and the link to that.

Now, once you have done the assembly, you need to assess the quality of the assembly. As we have done all throughout, we have checked the quality of the read, the quality of the mapping, and here we want to check the quality of the assembly. We very briefly talked about this assembly quality when we were talking about transcriptome assembly. So, similarly, when you are doing genome assembly, you would also have to look for quality. And one of the matrices that we defined earlier is called the N50 metric, and the explanation for N50 is very simple. So, let us say the N50 value is x. This means that at least half of all assembled contigs are of length x or higher.



- N50 metric

  N50 value = x

  - at least half of all assembled bases from contigs of at least length ‘x’

So, if x is, let us say, 5 kB, it will say that at least half of all assembled contigs will be greater than 5 kB. So, this is how we actually look at the data. So, if you want this value to be higher, that would mean you have managed to get close to the actual chromosome or genome. So, we also look at contig numbers, and we want these numbers to be as small as possible, and in the ideal world, we want them to be close to the chromosome number.

And in addition, we also check for genomic coverage and genomic completeness. So, how do you check this? So, from other experiments, we have some idea about the genome size number of chromosomes, etcetera. So, we can utilize that information and see right how much of that genome size is covered by these reads that we have or the contigs that we have generated, right how much of the full genome we have covered with the reads, and also the coverage right how many reads on average we see for each location of the genome. So, these are very important parameters and they have actually been implemented in different tools. Again, I will just simply mention these tools. You can find the links, and you can also go through the papers and utilize them for your research. Finally, the final step after we have checked the quality is something called genome annotation.

So, you have derived a new reference sequence, right, but then it is simply just the basis, right, ATGC, etcetera. So, what we need to do now is add more information, add something, or associate these bases with genes, exons, etcetera, and this process is called genome annotation. So, there are two types of annotation that you can do. So, the first is structural annotation, okay? So, this is about identifying structural elements of the genome, for example, genes coding regions, exons, or introns, and this you can do through something called gene prediction.

You can, of course, check closely related species or organisms and see whether there are some sequences that resemble genes from those closely related species. So, you can do this gene prediction, or you can also combine this analysis with transcriptome analysis. So, the advantage is that with the transcriptome, you know which regions of the genome are transcribed. So, you can now start identifying the gene regions—the coding regions—exons and introns, which we can do very easily if you also combine genome data with transcriptome data. The second type of annotation we do is called functional annotation. This is actually much harder because what you want to do is infer gene functions.

So, we have now identified some genes, but we want to infer gene function, and we have to do this through comparative analysis because the experimental determination of function is very difficult, tedious, and time-consuming. So, we would have to do this through comparative analysis with related genomes to see if similar genes exist in other organisms, and if we know the function of those genes in the related organisms, we can then say this gene also has a similar sequence. So, it is likely to perform the same or similar function. Alternatively, you can also do domain predictions. So, for specific functions, you have specific protein domains, and if you see the presence of certain domains in certain genes, you might be able to predict right away that this gene is probably associated with this kind of function.

So, here are the references for this class, and to summarize, we have extensively discussed deep brain graph assembly and have seen that it does not require any comparison for overlaps. So, it builds this overlap graph or multigraph using the k minus 1 mer in the data, and then we can identify whether there is an Eulerian path, which enables the derivation of the reference sequence. Of course, we also talked about exceptions. So, there are different scenarios where we cannot find an Eulerian path or where there could be multiple Eulerian paths, and we have talked about assembly quality assessment because once we have done the assembly, we want to check how good the assembly is. The tools that we have described or the algorithms that we have described, as you can see, are not perfect, but there is still a lot of scope for improvement. And then finally, we talked about genome annotation very briefly. This is for the identification of structural and functional elements, and this actually follows after you have done the assembly, which actually completes the whole process. Thank you very much.