Next Generation Sequencing Technologies: Data Analysis and Applications

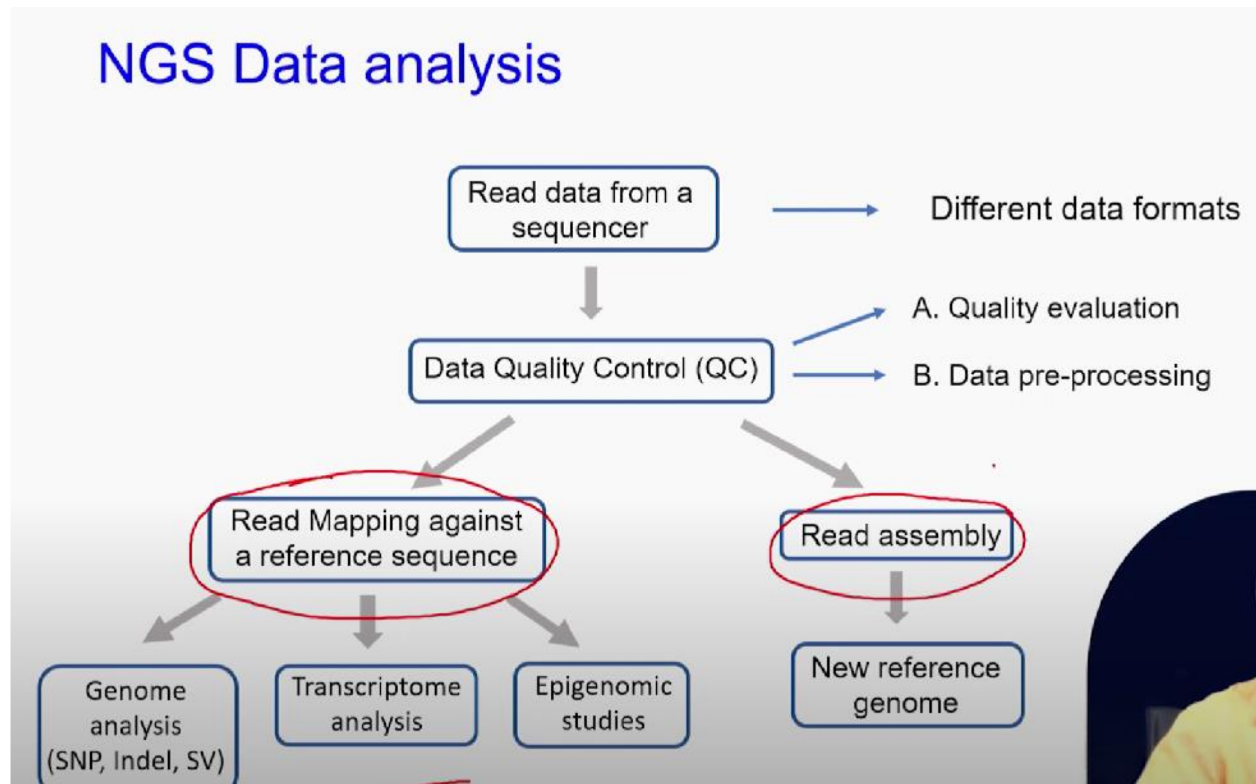
Genome Assembly

Dr. Riddhiman Dhar, Department of Biotechnology

Indian Institute of Technology, Kharagpur

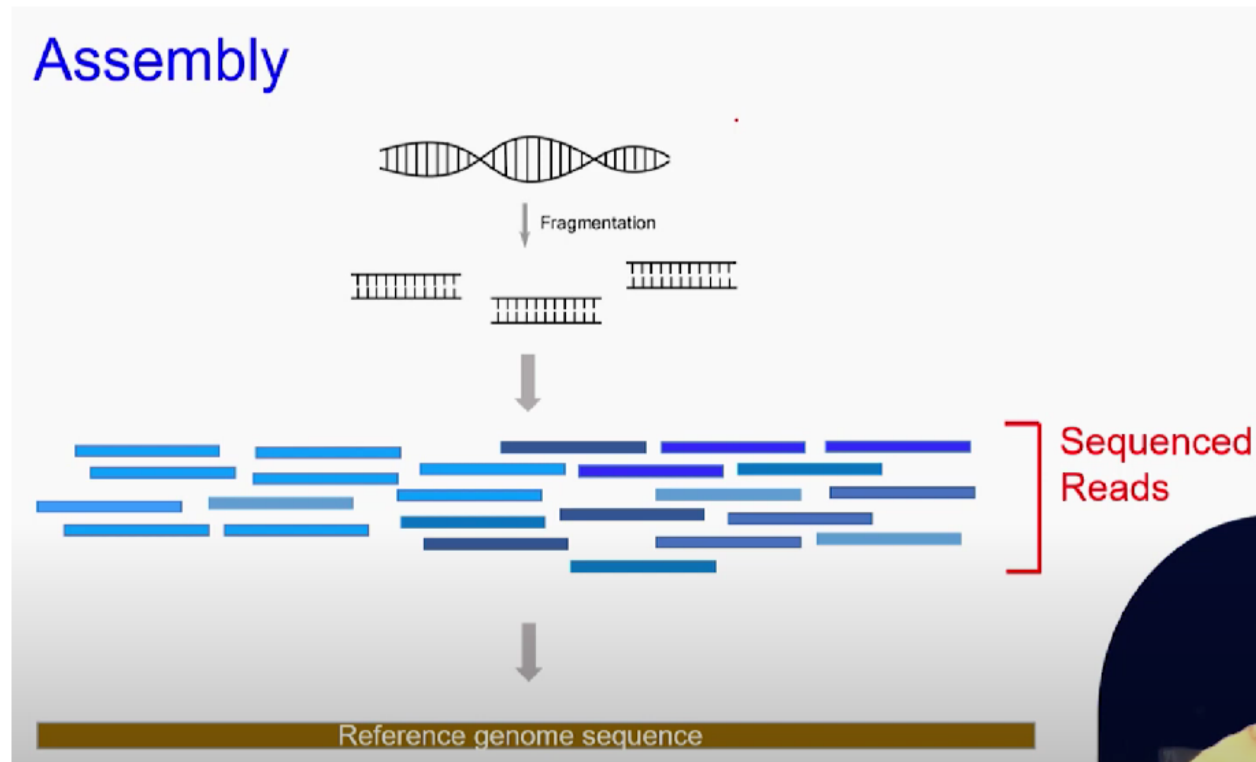
Good day, everyone. Welcome to the course on next-generation sequencing technologies, data analysis, and applications. In the last few weeks, we have talked about a lot of applications of next-generation sequencing, and these revolved around single nucleotide polymorphism analysis or copy number variation analysis and then we talked about transcriptome analysis. So, we analyzed RNA-seq data to actually compare gene expression levels across different samples from different conditions or different tissue types, etcetera. So, all these analyses rely on the reference genome, ok? So, we talked about mapping against the reference genome in the case of single nucleotide polymorphism analysis, or in the case of RNA-seq data analysis, we are also mapping against a reference genome, right? So, that required that reference genome. Now, the question is: how does this reference genome come about? Where do you get this reference genome from? So, this is what we are going to talk about this week, right? So, the topic would be genome assembly. So, how do you take next-generation sequencing data and actually build those reference genomes? So, that is what we will start with today. So, the concepts that we will cover today revolve around the genome-for-genome assembly problem. So, we will first talk about the genome assembly problem, right? So, how is it different from the mapping problem? So, we talked about mapping, whether it is RNA-seq or targeted re-sequencing, right? So, there was this mapping, and we talked about algorithms that actually help in the mapping process. So, similarly, we have this genome assembly problem, and especially the genome by genome assembly, right? So, we are building this genome from scratch, right? So, we do not have any comparisons or anything. So, this is a very difficult problem. So, we will first elaborate on this problem, so that you understand how difficult this problem actually is. And then we will talk about some ways to address this assembly problem, right? So, how do you go about assembling genomes from the reads that we get from the next-generation sequences? So, we will talk about one such approach today; it is called the shortest common super string approach, or SCS approach, and then in subsequent classes we will talk about other approaches. So, the keywords that will come across today are overlaps and directed graphs.

So, let's begin, right? So, with the flow chart that we started with, ok. So, here we are. We have the read data, and we do the data quality control.



So, this step is common across all the analyses. Then we talked about the left part mostly, right. So, we talked about this read mapping and then genome analysis and transcriptome analysis. We have not talked about epigenomic studies yet, which we will cover in the last week. So, the other part that is remaining is the read assembly part, right? So, this is part; this is the part that we have not discussed yet, ok? And this is what we are going to talk about. So, the part in green is complete, right? So, we have discussed those in this course. The part that we are going to discuss today is the read assembly part, and then we will follow this up for the subsequent classes, ok? So, let us now understand the assembly problem and why it is a challenging problem. So, here is the simple schematic that shows which is showing DNA, which is first fragmented into small fragments, and then they are processed according to this adaptor ligation, amplification, etcetera, depending on the platform. We have discussed those numerous times, ok? So, I am not going to go to that. So, once we get these fragments, we process them through NGS, and what we get are the sequence reads. So, these reads are shown in blue here, ok? Now, once we have these reads, the assembly

means that from these reads, we would have to actually generate the sequence of this DNA that we started with. We have no idea about the sequence of the DNA that we started with, and we want to generate that sequence. The order of these sequences is right. So, each read probably will have some unique sequence, right, and how they occur in the genome, right.

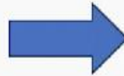


So, we have to kind of order them right in and then find out the full genome sequence, ok? So, that is the problem, right? This is the assembly problem. We have to get the reference genome sequence. Of course, I am showing this as one single stretch of DNA.

It is not right. If you are talking about eukaryotic or mammalian genomes, right, we have multiple chromosomes. So, the idea is to get those chromosome sequences from this read data. So, this is the problem, ok? Now, let us understand the challenges, right? So, with a very simple analogy that we are all familiar with, So, we must have played these puzzles, right? So, these pieces, right, that you have to put together to get a picture, right, to generate a nice picture, ok? So, these are the pieces of the puzzle. So, you can imagine these are like reads, which you would have to put

together. We have fragments of this genome, which you have to put together to generate the full genome sequence.

Assembly



But without any reference image!

Now, when you put them together, this will look something like this, ok? But then one thing you should remember, right? When you do these puzzles, you have a reference picture somewhere, right? So, you look at the picture and say, Ok, this seems to fit with this, this seems to fit with that, right? So, then you can put this together. When you are doing the assembly, you do not have that reference picture, right? So, you have to somehow put them together so that they make sense, right? So, you have a rough idea of what you are going to see, but you do not have any reference images, right? So, you can now understand how difficult this problem is, right? So, if I give you, let us say that there are even 1000 pieces of a puzzle, and I say, ok, assemble them so that they make some sense, right? They make some nice pictures, but I am not going to give you any reference picture or reference image, ok? So, that is what the assembly problem is.

We have no idea about the reference genome sequence, ok? We have some sort of number of chromosomes etcetera, as we can guess, right? We can know right by other methods, but apart

from that, we will probably have no idea about the sequence of the genome, ok. So, without looking at any picture or any reference, we have to put them together so that they make some sense, ok. So, this is what the assembly problem is, and that is why it is incredibly challenging, ok? So, let us now work with very small DNA sequences and then try to understand this problem and see how we can solve it. So, once we go into this understanding, right, once we understand what this problem is about, we can think about solutions, right. So, how can we approach this problem and try to solve it? Okay, so let us take this example where we have this reference DNA; it can be any DNA, right? So, I have just taken one sequence, and we take this sample, right? So, when we take this sample, we have this DNA in multiple copies from multiple cells, right? So, let us say you are working with a population of cells, and you extract genomic DNA and then sequence it. So, in a sense, you have multiple copies of the same genome sequence. Okay, maybe with some minor differences here and there, but most of them are identical. So, you generate these multiple copies. Here, I am showing only four copies, right? So, you can imagine this can be millions, depending on the number of cells you are working with, and then fragment, ok? So, the fragmentation process will generate random fragments, right? So, it will fragment at random places, mostly unless you are using restriction enzymes for digestion. So, any other, such as physical fragmentation, is right. So, those will generate random fragments, ok? So, I have shown these fragments by these spaces in between, right? So, we have these for this copy.

Assembly

ATAGGACTAGCAGGAGCCTAGAGAGAGACGGCA

ATAGGACTAGCAGGAGCCTAGAGAGAGACGGCA

ATAGGACTAGCAGGAGCCTAGAGAGAGACGGCA

ATAGGACTAGCAGGAGCCTAGAGAGAGACGGCA

Input

Fragmentation

ATAGGACTAGCAGGAGCC TAGAGAGA GACGGCA

ATAGGA CTAGCAGGAGCCT AGAGAGAGACGGCA

ATAGGACTA GCAGGAGC CTAGAGAGAGACGGC

ATAG GACTAGCAG GAGCCTAGAGAG AGACGGCA

Let us say we have this fragment: one fragment here, then the second fragment, then the third fragment, and it is very unlikely that you will get fragmentation at the same place for all DNA copies. So, that is why you see different types of fragments. So, what we will do is take these fragments, process them through this adapter ligation step amplification, and maybe there is some sort of size selection where we might remove very short or very long fragments depending again on the platform that we are using.

Assembly

ATAGGACTAGCAGGAGCCTAGAGAGAGACGGCA Input DNA

ATAGGACTAGCAGGAGCC
TAGGACTAGCAGGAGCCT
GGACTAGCAGGAGCCTA
GACTAGCAGGAGCCTAGAGA
CTAGCAGGAGCCTAGAGA
CAGGAGCCTAGAGAGAGACG
GGAGCCTAGAGAGAGACGGC
AGGAGCCTAGAGAGAGACGGCA
GAGAGAGACGGCA

Sequenced Reads

So, there will be an appropriate size, and then we will sequence these fragments, ok? So, now, what I will say is, let us say we generate these sequence reads and we need to generate this input DNA sequence, ok? Now, if I arrange these sequence reads like this in this order, right, we can see, ok, this read kind of overlaps with this one, and if I arrange these reads in order, it is easy to generate this assembly, right? So, it is easy to generate this input DNA sequence, right? So, if you look at this and say, Ok, yeah, that is it. So, this should be the reference sequence or the input DNA sequence that we started with.

Can we assemble this?

ATAGGACTAGCAGGAGCC

GAGAGAGACGGCA

TAGGACTAGCAGGAGCCT

GGACTAGCAGGAGCCTA

GACTAGCAGGAGCCTAGAGA

CTAGCAGGAGCCTAGAGA

CAGGAGCCTAGAGAGAGACG

GGAGCCTAGAGAGAGACGGC

AGGAGCCTAGAGAGAGACGGCA

Now, the problem is, of course, that the reads will not be ordered like this, right? When you get from the sequence, ok, and they will look something like this, ok? Now, when you have this kind of order, right, and this kind of random fragments and random sequences, how do you actually generate that input DNA? And even with this small example, where we have only a handful of reads, as you can see, this becomes quite a challenging problem, right? It is not so trivial. Again, of course, there are ways you can now start to think about, right, how we can solve this issue. So, you can imagine that with millions and billions of reads, this will become very, very difficult as well as time-consuming. So, let us think about some approaches, right? So, the question is, how can we do the assembly? Let us start with a very simple example that I showed you, and then think about, right, how we can actually extrapolate that to situations where you have millions or billions of reads, ok? So, there are three types of assembly algorithms that we will discuss, right? So, these algorithms have been designed to do this task precisely, so to generate this reference sequence

from the read data, ok. So, these algorithms are employed for this purpose. So, the first one we will discuss is called the shortest common superspring approach, or SCS approach. The second one is the overlap layout consensus, or OLC approach. The third one is the Bruijn Graph Approach or de Bruijn Graph Assembly, ok? So, we will discuss each of these approaches one by one, ok? So, we will start with the shortest common superspring approach, ok? So, this is derived from this idea in computer science, right? So, you have this spring, ok? If you are given some strings, you can have superstrings, ok.

Assembly algorithms

- Shortest common superstring (SCS) approach
- Overlap-Layout-Consensus (OLC) approach
- de Bruijn Graph (DBG) approach

So, what are superstrings, or what is a superstring? So, it is a string, right, that contains all given strings as substrings, ok? So, we will take an example, and that will be easy to understand, ok? So, let us take this example here, a very simple one. We have four strings; let us say we have given four strings, ok, so rain, day, cloud, and sunny, ok. Now, for supersstring, right, this will contain all these strings, ok, and these will then be called substrings of the supersstrings, ok.

Now, you can generate superstrings very easily if you just concatenate, right? If you just put one after another, right? So, that is an easy process, right? So, you can simply put them together, and you can see that you can find this string rain in this superstring. So, this is the superstring here.

You can see you can find rain here; you can see you can find day here, right; you can find cloud here; you can find Sunday here, right.

So, this is an easy way to generate this supersstring, ok? Now, what is tricky is that you can probably notice it is called the shortest common supersstring or the shortest supersstring, ok? So, the shortest supersstring is the supersstring, right? It is the shortest one that contains all given strings as substrings, ok? So, out of all the superstrings that you can think of, this will be the shortest one, ok? So, this is becoming tricky now, right? What is the shortest one? Okay, and that is likely to be a difficult problem. Now, let us go back to the same example, ok, where we have these strings: rain, day, clouds, Sunday, right? This is the superstring, right? The shortest common superstring you can see for this simple problem, right? For a simple example, you can see this rain, clouds, Sunday. So, why is the day? Because the day is already contained in Sunday, right? So, we do not need to add another day here, right? So, we can have this only on Sunday, and that will already include the day, ok? So, this is the shortest common superstring. So, you can now imagine, right, we have given different combinations of words. You need to find these superstrings shortest ones, an this might not be very trivial as you expand this number of strings. So, as you increase the number of strings, this might be a difficult problem, right? So, you can think of it in terms of reads now, ok? So, these strings that are given are the reads, ok? In our case, these are the read data, ok? Now, from this read data, if we can find out the shortest common superstring, we can perhaps get close to the reference sequence, right? So, that is the idea; the reference sequence will contain all the reads that are generated, right? So, this is the idea with which we actually take the shortest common superstring approach, ok? So, let us now go into the shortest common superstring with reads, ok? So, let's again take a very simple example with 4 reads of length 4, right, 4 bases. So, you have A C G A, A A A T, C C G C, and C G A T, ok? So, the superstring if we just define, we can concatenate, we can just put one after another, and that will be the superstring. What will be the shortest common superstring? It is a bit tricky now.

We can probably figure out one thing: this part is common between these two reads, A C G A and C G A T, and perhaps you can have this A C G A T that will actually contain both of these reads, ok? And for the other two reads, you have to have these separate fragments, because they are unique. Now, as we increase the number of reads, these kinds of situations will be more common.

Shortest Common Superstring with reads

Reads obtained: ACGA, AAAT, CCGC, CGAT

Superstring : ACGAAAATCCGCCGAT

SCS : ACGATAAATCCGC

Basic principle of assembly

Reads coming from the same region of the genome would show sequence overlap

```
ATAGGACTAGCAGGAGCCTA

ATAGGACTAGCAG
        AGCAGGAGCCTA
```

Here, we have just one situation where we see this sharing of sequences between these two reads, but as we increase, as we go into millions and billions, we start seeing a lot of these sharings, or overlaps, as we call them. So, this will make the problem more challenging, ok? Now, how do we

actually find these overlaps? So, where do we start? So, as you can probably guess now, we start with the overlap part, right? So, if we have reads coming from the same region of the genome, they are likely to show sequence overlap, right? So, that is the idea, ok? So, if you take a fragment from this genomic region and let us say you have two reads that are coming from the same region, right, they are likely to show overlap, and here in this example, you can see this overlap here, A G C A G, this part is overlapping between these two, ok. So, with this idea, we can now continue, right? So, what we do is represent reads in a directed graph that actually tells us about the overlap, ok? So, which we also call an overlap graph, because this directed graph tells us about overlaps between the reads that we have got from the sequence, ok? And as part of the graph, we have nodes, right, or vertices, right.

Reads are represented with a directed graph

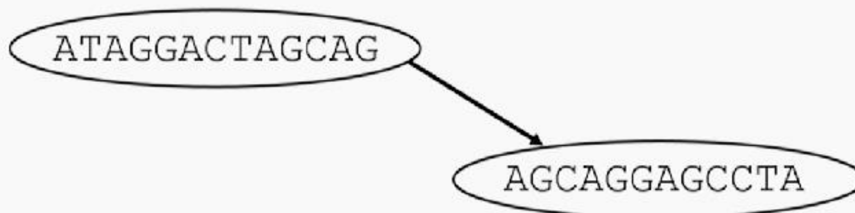
Node or vertex - Each read is a node in the graph

Edge – Two nodes (or reads) are connected by an edge if they overlap

Direction of the edge?

- Suffix of read 1 showing overlap with prefix of read 2
- direction of edge from read 1 to read 2

As we have discussed very briefly, like the graph, what you have in a graph. So, we have nodes and edges, right? So, each node is a row read, right? So, a read is represented as a node in the graph, and then two nodes or reads will be connected by an edge if there is overlap in sequence between these two, ok? Now, this is a directed graph, which means you also have directions, ok?



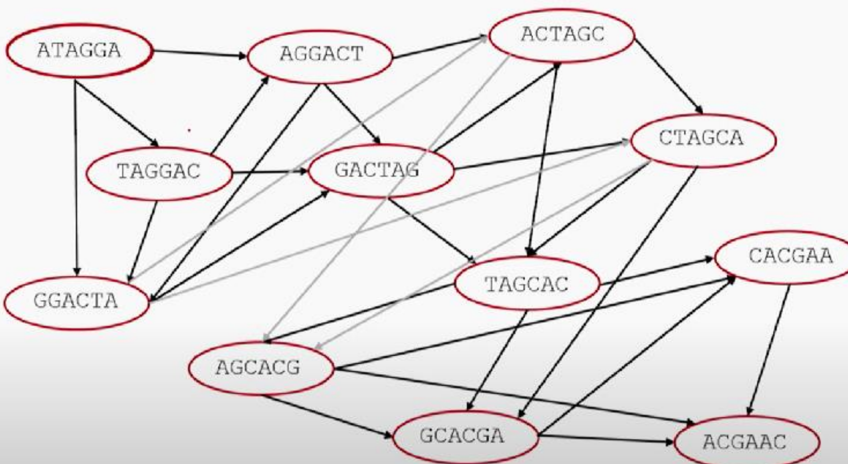
So, you have arrows from one node to another. So, how do you decide these arrows or these directions? So, that should go from node 1 to node 2, right? Assuming node 1 and node 2, they are showing overlap. So, would the arrow go from node 1 to node 2 or from node 2 to node 1, ok? So, that is something we also need to decide, ok? How do you decide that, right? How do you decide the direction of this edge? So, this is decided by the following, right? So, if you are seeing overlap between read 1 and read 2, and the suffix of read 1 is showing overlap with the prefix of read 2, then we will add the direction of the edge from read 1 to read 2, ok? So, read 1 would be the starting node, and read 2 would be the end node, ok, for the arrow. So, that direction will be from 1 to 2, ok? So, let us take this example, right? So, where we have seen this overlap between these two reads, ok, and for the first one, this part is the suffix, right? This is towards the end, right? So, again, you remember these definitions of suffixes, right? So, we talked about the suffix tree, etcetera. So, here is the suffix of the first read, which is showing overlap with the prefix of this second sequence. So, that means this will be the node from which the arrow will start, and this will be the node from which the arrow will end, ok? So, that is why you have to see now the direction of this arrow here, ok? So, what you need to do is we need to do this for all pairs of reads that are present in the data, ok? So, we have over a set of reads now, ok, and this is the input DNA again.

Let us say we have this very nice pattern of reads. We have got all the reads that are possible of a certain length. So, here we are taking length 6 spaces, ok, just for our example, and we have got this nice overlap; you know, these are the overlaps, etcetera.

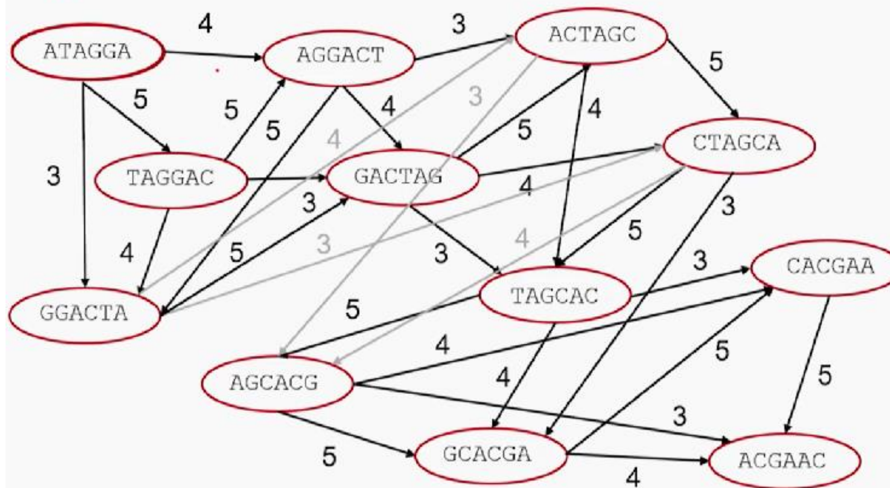
We will talk about how we actually find overlaps between reads in reality. What are the methods? I will just briefly mention them probably in the next class, and then you can utilize those methods to actually find the overlaps between the reads. So, what we do is start with this pairwise comparison between reads for overlap, ok? So, what you do is compare this read 1 with read 2, ok? Then you go to read 1 and 3, right, and you go on, right. You continue this for all the pairs that you have in the data. So, as you can imagine, this will take time, ok? As you have millions of billions of reads, all pairwise comparisons will take a long time, ok? So, of course, these overlap graphs are time-consuming, ok? So, you have to build them in case you have millions of billions of reads; then it will take time, ok? So, again, when we check for overlap, we also check for this prefix and suffix part, right? So, which reads where the overlap is—whether it is in the suffix part or the prefix part—because this will decide the direction of the edge in the directed graph, ok? In addition, we also have something called the cutoff length for overlap. So, we do not say just if you

have just one base overlap; we do not consider that as an overlap or two base overlap. We need a certain amount of overlap, right, just to avoid any spurious overlaps. o, it overlaps just by chance. One base overlap will happen just by chance, or a few base overlaps will happen by chance. So, you need to decide on a cutoff length. So, as you will see in these examples, as we go on, we will choose this cutoff length, ok, and then we will draw the directed graph or generate the directed graph with all the directions or the arrows, etcetera, ok. So, if we continue this process, we will see we end up with something like this, ok.

Directed graph



Cut-off length for Overlap : 3



So, here I have taken this cutoff length for overlap as 3. So, we need at least 3 base overlaps between 2 reads, and then again, we know the suffix and prefix parts for each read, and depending on that, we add these arrows. So, you can derive this yourself, and I encourage you to do so with a pen and paper. It is pretty simple. If you just sit around and do these pair-wise comparisons, right, for example, these two, you can see this will be the direction of the arrow, and then you see the overlap is about 5 bases, ok. So, you can do this for all of them, and then you can simply add these arrows for these overlaps, ok.

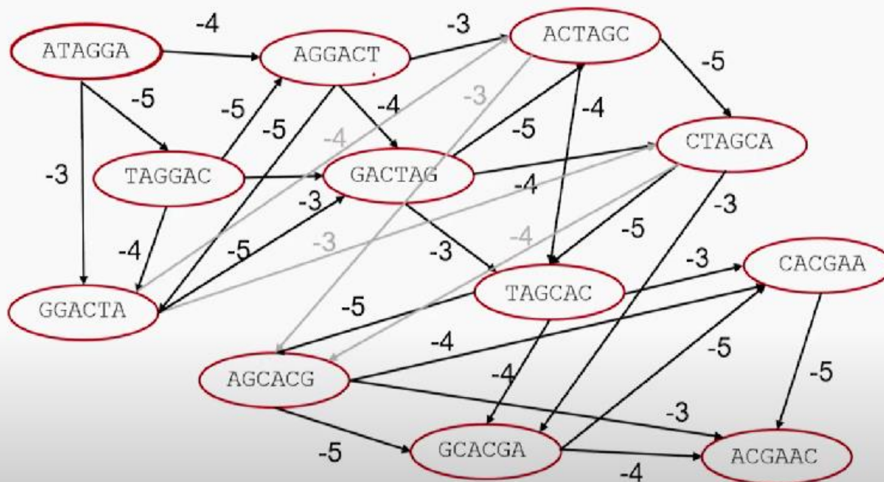
And also, we have this, where you can see two different colors. They do not; they are not actually different. They are just showing for visualization purposes, ok? So, we can see them better with these nodes, ok? Some of these arrows are going through the nodes, ok? So, it looks quite complex, right? For a small number of readers, it is already quite complex. So, as you can imagine, if we work on a large number of reads, this will become really complex, and it will be a huge graph, ok? Now, let us move on with this example. What do we do after we have built this directed graph? So, we also add the edge weights to the graphs, ok? It is not just this directed graph. We also add edge weights to the graph, ok? So, what are these edge weights? These are actually given as lengths of overlap, ok? So, we can add these numbers now to this graph which shows how many bases are

overlapping between these two reads, ok? So, the nodes you remember these are the reads. So, this is lead 1, etcetera, right? So, we have all the reads represented as nodes, and you have the edges, and then you have the edge weights, ok, showing the number of bases that are overlapping between these two reads that are connected by an arrow, ok.

So, what is the purpose, right? And what are you going to do with this directed graph, ok? So, again, you can derive this quite easily, right? So, you can try to derive this from the information, right, of the read sequences that we have just seen, and then you can derive this yourself, ok, with the simple idea of how the nodes will be connected, the direction of the arrows, and the edge weights, which will be decided by the overlapped length, ok. So, as you can now see, right, we will utilize this directed graph to actually get to the reference sequence, ok. So, can we? Now, the idea in the shortest common superstring approach is that we can determine the reference sequence or derive the reference sequence from this overlap graph, ok? Now, what do we have to do if we want to do that? What do you do? What are you going to do? So, how can we find the shortest common superstring? So, the idea is that if we can find a path that traverses through every node in this graph exactly once and maximizes the total scope, ok.

So, you can actually traverse through these graphs, ok? So, you can maybe start from here; you can go in this direction, then go in this direction, and so on, right? You can see that there will be multiple such paths, right? So, you have so many options at every node, ok, and you can take any of those. What the SCS approach tries to do is find a path that traverses through every node exactly once and maximizes the total scope.

How can we find the SCS?



Cut-off length for Overlap : 3

How can we find the SCS?

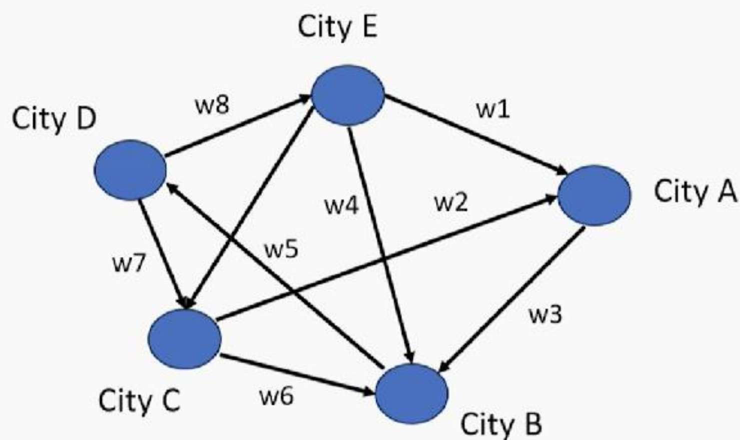
- Cost = - (overlap)
- Find a path that traverses through every node exactly once and minimizes the total cost
- Equivalent to *Traveling Salesman Problem (TSP)*

So, why every node exactly once? Because we have read, right? So, we are using this one read only once, right, not multiple times, and the total score is calculated by the sum of the edge weights, right. So, if you are traversing through, let us say, weight it as edge weight 4, then edge weight 5, you simply add this, right, 4 plus 5 plus and so on, ok, and find you get the final scope, ok. So,

how do you actually find this path? So, what do we do? We will transform this score into something else, right? So, we can think about something called cost instead of these edge weights or scores, and we can define cost as the minus of overlap, right? Just change the sign, ok? We are not doing anything; we are just changing the sign, ok? And the graph will look the same. What we have done is change the sign of these edge weights, right? So, they are negative now, ok? And what we have to do is transform the problem, right? So, this means we have to find a path that traverses through every node exactly once but also minimizes the total cost. Instead of maximizing the total score, now we need to minimize the total cost, right? So, we want as negative a number as possible, ok? So, that problem is transformed, and you will see why that is the case because we see now that this has resemblance to something called the traveling salesman problem or the TSP problem. This is a very famous problem in computer science, right? So, this is a travel salesman problem, and we know how difficult it is to solve it, etcetera.

Shortest Common Superstring (SCS) approach

Traveling salesman problem (TSP)



If we do not consider the weights, it becomes *Hamiltonian path problem*

So, what is the traveling salesman problem, right? So, you can think of this scenario, right? The name probably tells you that already, right? So, you can imagine an individual traveling through

multiple cities, ok? So, again, this is a graph of cities, right? So, cities A to E and these cities are connected by an edge, ok, because there are, let us say, flight connections or train connections between the cities, ok. So, for this individual, what he or she would have to do is travel through each city exactly once, ok, and also minimize the cost of travel, ok.

So, you have to find a path that will traverse through every node exactly once but will minimize the cost of travel. So, the cost here is given by these W's, right? There are weights on these edges, right? So, this cost would have to be minimized, and the individual would have to travel to every city exactly once.

You can see this striking similarity with our problem, right? So, we want to traverse through every node exactly once, and we also want to minimize the cost. And this problem is known to be something called NP-hard, ok? So, without going into a lot of details about what is NP-complete or NP-hard, it will be simple to say that this will take a long time as we start adding more nodes to the graph, ok? So, for a small number of cities or a small number of streets, we can still solve it within a reasonable amount of time, but as we start adding more nodes to that graph, this will increase very quickly, ok.

So, if the time required to solve that problem is quite long, ok. So, you can imagine that in our case, when you have millions of billions of people in reach, this is going to take a long time. We can think of this; we can simplify it, right. So, if we do not consider the waste, if we just say we just want to travel through every city exactly, once you can find these solutions, you can see that there are multiple ways to do this, and then it becomes something called the Hamiltonian path problem, ok? So, again, this is a famous problem in computer science, and it is known to be NP-complete.

So, again, we are not going to go into the details of this NP-hard or NP-complete. What it means is that it is going to take a long time as we start adding more nodes to the graph, ok? So, what we see is the shortest common superstring approach, right? So, it is going to be quite time-consuming as we have more nodes in the graph. So, the question now we are left with is: how do we determine the SCS, or shortest common superstring, from the overlap graph in a reasonable amount of time?

So, this is something we will talk about in the next class, and we will talk about other approaches in the following classes. So, here are the references for this class. So, to summarize, we have talked about the de novo assembly problem, and I have discussed why this is a really challenging problem, what the reason is, etcetera. And we have discussed the SCS approach of the shortest common super string approach. So, this starts with a pair-wise comparison of overlaps between reads. You will see other methods; one of the other methods that we will discuss also utilizes this overlap graph as the starting point. And then reads are represented in the directed graph with edge weights, which are again determined by the length of the overlap.

The direction is also determined based on the overlap region, right? So, whether the overlap is in the suffix or in the prefix, that will determine the direction of the arrows. And finally, we have discussed that determining the shortest common superstring from the graph is equivalent to the traveling salesman problem and is an NP-hard problem. Thank you very much.