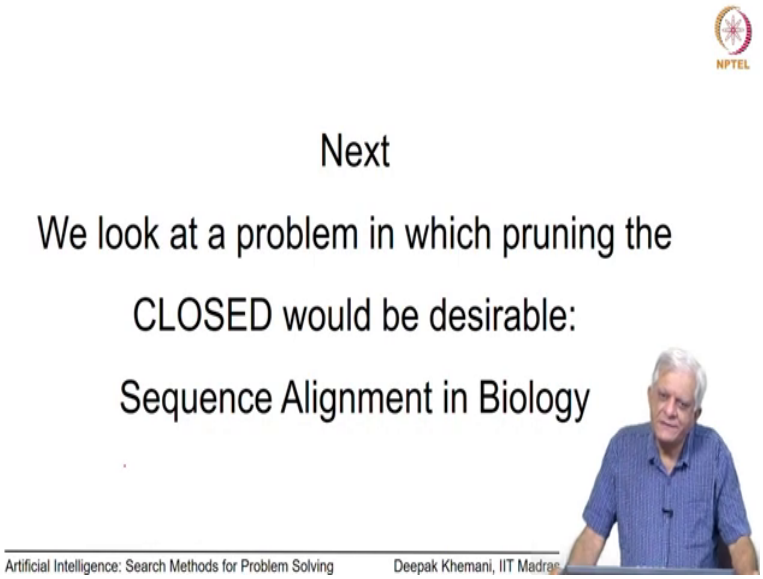
Artificial Intelligence: Search Methods for Problem Solving
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Chapter – 05
A First Course in Artificial Intelligence
Lecture – 44
DNA Sequence Alignment

(Refer Slide Time: 00:14)

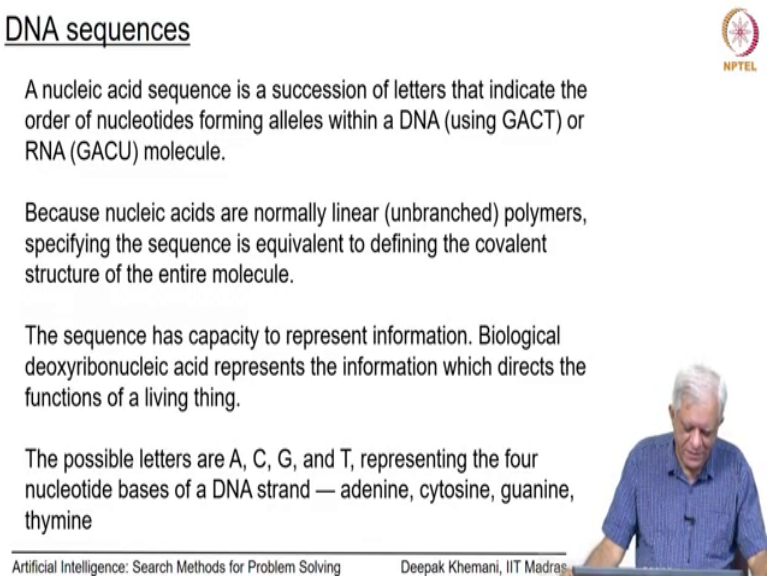


So, welcome back in our study of A star algorithm. The last thing we studied was the monotone property or the consistency property on the heuristic function, which said that the heuristic function underestimates the cost of every edge in the graph, and we had said that the consequence of that is that the nodes that we have to add to closed do not have to be updated again because they already have the optimal path found.

And therefore, if we can manage to not keep them it and devise algorithms which do not need closed, then it would be a considerable saving on space. So, let us as I said towards the end in the last session, let us take a deviation from our algorithm A star and look at a problem for them from the real world which comes from the world of biology which would benefit tremendously if we had algorithms which required less space, because the people who work in biology they work with huge amounts of data.

And one of the problems that we are looking at or about to look at in the sequence alignment problem can be posed as a search problem and that is what we will see in this session.

(Refer Slide Time: 01:35)



So, just a bit of talk on biology, we are talking about DNA sequences. And essentially what happens many times is that the genome, for example the human genome is a huge, huge, huge string of characters. And very often when people are studying DNA and things like that, they

manage to identify small sections or subsections and so on. And for various reasons, they need to align the data that they get from one source to data from another source. And this problem has been for less sequence alignment problem.

So, just a little bit of biology before we move on, a nucleic acid sequence is a succession of letters that is important for us that we are not going to be handled with matter organic matter, we are not talking about cells and things, all that kind of stuff, we are simply talking about sequences of letters.

So, a nucleic acid sequence is a succession of letters that indicate the order of nucleotides forming alleles within a DNA or within an RNA as the case maybe. We will look at DNA here.

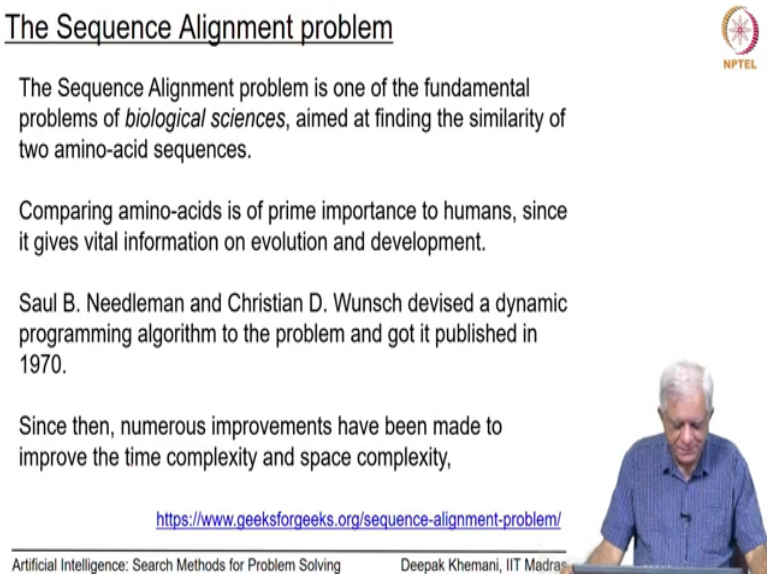
Because they are normally linear, they do not have branches polymers, specifying the sequence is equivalent to defining the covalent structure of the entire molecule. So, how many bonds? So, remember that carbon has 4, has a valency of 4, and it has 4 bonds, and you know in a molecule all the valencies must be balanced and so on. In the case of DNA like molecules, if we specify the sequence, we have also specified how many bonds each atom has.

The sequence has the capacity to represent information, obviously, we know that we know that you know it is we say that it is in her genes or it is in his genes and so on. And the DNA is what you inherit from your parents, and it is the DNA which decides how your body is built up, and how you grow, and what kind of hair color you have and that kind of all that kind of stuff we know it represents information.

And DNA which is deoxyribonucleic acid represents the information which directs the functions of a living being. It, it is like a program, which tells our body how to grow essentially. So, it is clearly and very important from the information processing point of view as well essentially.

The letters that we are interested in are of A, C, G, T which stands for adenine, cytosine, guanine and thymine. And we will just henceforth, refer to them as ACGT. And what we are talking about DNA strands are sequences of letters using these four characters A, C, G, T, which are nucleotide bases essentially.

(Refer Slide Time: 04:36)



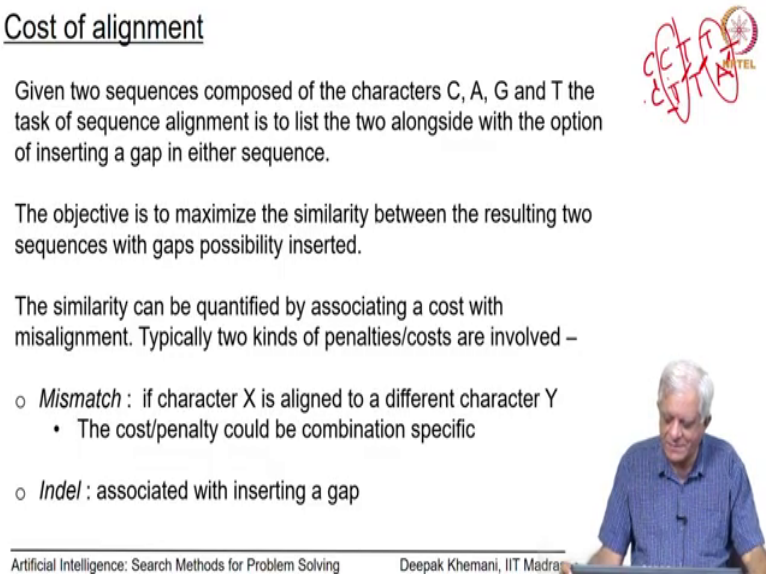
So, what is a sequence alignment problem? The sequence alignment problem says that if you want to find the similarity between two amino acid sequences, what you need to do is to align them together in some sense lay them side by side, and see how you can align them, so that the matching is the best essentially. And as we have said that this is the fundamental problem which comes from the biological sciences.

And the aim is to find sequences between similarity between two such sequences amino acid sequences. It is of prime importance to humans since it gives vital information about on

development and evolution. And we know that nowadays people talk about having the an that you know we have sequence the genome of this creature or that creature and so on, and this is all part of that work essentially.

Now, quite some time ago in 1970, Needleman and Wunsch, they devised a dynamic programming based algorithm for solving the problem of sequence alignment. And it has been used since then by people who work in biology. But subsequently there were improvements and we are going to look at one of those improvements to improve the time and space complexities essentially, in particular we are going to be focusing on space complexity as I have been saying.

(Refer Slide Time: 06:09)



So, let us talk about what do we mean by this similarity, what do we mean by matching, and how can we associate and cost with it. So, given two sequences composed of this character C,

A, G, T as we have said the task of alignment is to list the two next to each other to list the two alongside with the option of inserting gaps in either sequences.

So, in the interest of maximizing similarity or minimizing cost as the case may be which are equivalent in nature, we may find that it is easier to insert gaps between sequences. So, for example, if you have a sequence C C T and another sequence C T T, let me add more C to that, or let me add the character A here, then maybe one possible alignment is to align the T with the T, thus C with the C, and the A, so you must need to insert a gap here to match with the A and another gap here to match with the C maybe that is a better assignment.

So, we may want to insert gaps. We will denote gaps by underscores here. So, we can insert a gap here and insert a gap here, it might give us better alignment, so that is the idea of alignment where you may insert gaps. So, obviously, you can see that the number of possibilities will now rapidly grow. The objective is to maximize the similarity between the two sequences with gaps possibly inserted as we have just seen in this example.

The similarity can be quantified by associating a cost with misalignment. And typically there are two kinds of costs. One is a cost of mismatch. So, if a character X is aligned to a different character y, then the cost could be then we associate a certain cost with that essentially.

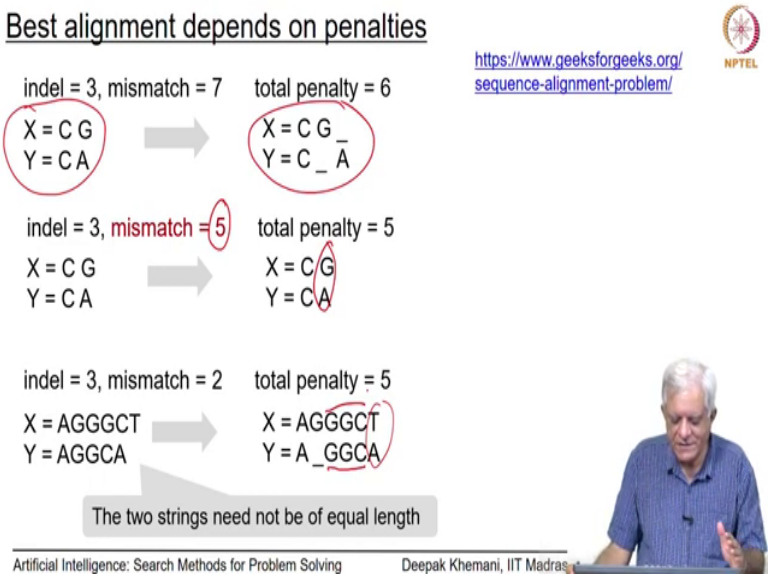
So, for example, if we had said that we will align this T with this A, and this C with this T, then we may have to pay a certain cost which would be added to the cost of the solution. Our task is going to be to find a minimum cost solution or a maximum similarity solution depending on which way we look at it, but both are two sides of the same coin in some sense.

So, how much should be the cost? The cost could be uniform, it could say that; if it if there is a mismatch, there is so much cost; or the cost could be combination specifically it could be that if you are mismatching C with T, then there is a certain cost. But if you are mismatching T with A, then there is a different cost.

So, that depends on how closely you have monitored, how closely you have formulated your problem, and what is the notion of similarity that you are looking for. We will see some examples as we go along.

The other cost is a cost associated with inserting a gap. So, if you are going to insert a gap, then obviously, you are changing doing something to the strings, and that should add to the cost essentially. So, we have two kinds of costs, one is the cost of mismatch and the other is the cost of inserting a gap.

(Refer Slide Time: 10:04)



Now, the alignment that is the best would depend upon what is this what are these costs that you are associating with, or what are the penalties you are associating with, either

mismatching two characters or inserting a gap. So, let us see a couple of small examples which I have taken from the site which is listed here.

So, on the left hand side is the input and that is the two sequences X and Y that we are want to align. It is a very small sequence. X just simply contains of C and G, and Y contains C and A. Now, if the gap insertion cost is 3 and if the mismatch cost is 7, then you can imagine that it makes sense to insert a gap.

And what you see on the alignment is that you have inserted two gaps, one to, in one, one to match with C and the other to match with A. And the total penalty is 6 which is because the gap insertion cost is 6. So, we have inserted two gaps. So, the penalty is 6, and the all of the cost of the solution is 6.

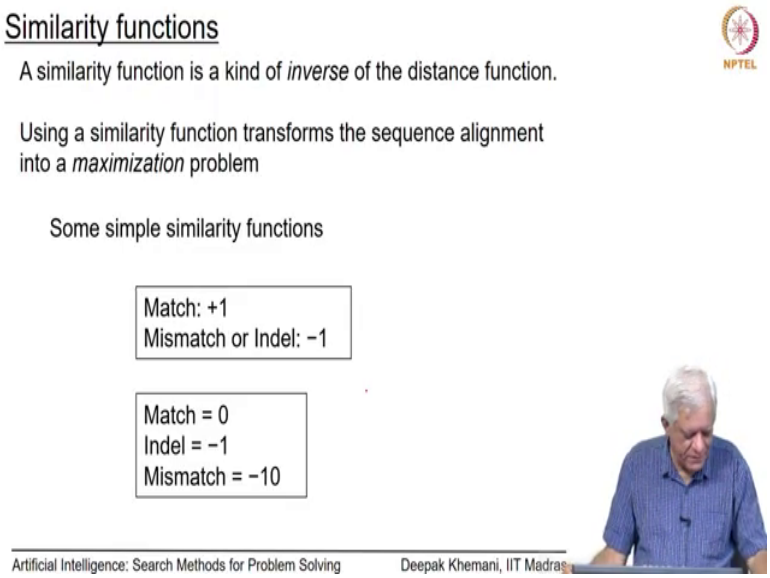
But if the mismatch costs were to be different, so for example, if the gap insertion costs or insert delete costs as it is sometimes called indel cost is 3. But the mismatch cost is lower, then you can see that leaving the two strings as it is gives us a lesser penalty that if we are inserted the gaps as above we were got a cost of 6, but without inserting the gaps and simply saying that ok, we let this G align with A and because the mismatch cost is 5, the total cost of the solution turns out to be 5.

So, one thing one should note is that the strings that you are aligning do not have to be of the same length essentially. You can also align strings which are of different lengths; simply by inserting gaps to make one string equal to in length to the other, and then the alignment would be complete. So, for example, if you have these two strings the upper string X has got 6 characters AGGGCT the lower string Y has got only five characters A GGCA.

And if you assume that the indel cost is 3 or the gap insertion cost is 3 and the mismatch cost is 2, then we will if we insert one gap to push this GC GGC to align with the GGC on the top, and if we are willing to tolerate this misalignment T of A, which should cost 2 remember gap insertion cost is 3.

So, the cost of this particular solution which is the best solution is 5 essentially. So, the task of sequence alignment is to basically align two sequences such that the cost as defined by the penalties is minimized essentially.

(Refer Slide Time: 13:11)



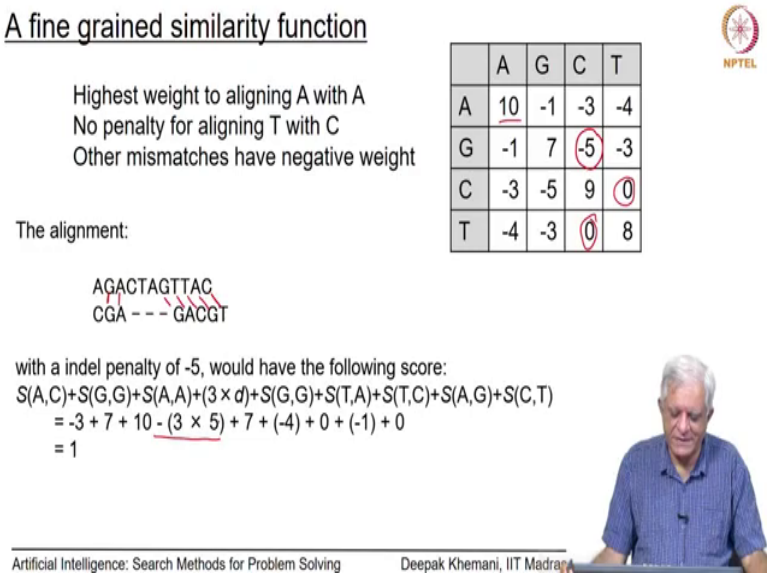
Sometimes we talk in terms of similarity as well because we are interested in knowing how similar the two things are. And the similarity function in general is a kind of a inverse of a distance function. The more far apart two entities are the less similar they are that is the basic idea essentially. In general of course, the similarity functions are scaled down to map in the range of 0 to 1. 1 means identical; 0 means totally different; and any value in between means some degree of similarity.

And we do not do that with distance function we keep the distance functions as they are. But you can see that one can always take a distance function and convert it into a similarity function without any loss of information.

But if we were to use similarity function directly, then our problem would become a maximization problem as opposed to a minimization problem that we are talking about we are talking about finding a shortest path or the least cost solution. Whereas, now we would say that find as the solution which has the maximum similarity between the two strings.

So, here are some similarity functions. You say that if the two characters match, then you add 1 to the similarity, and eventually of course, you will scale it down. If they mismatch, all if you are inserting a gap, then subtract 1 essentially. Or you could say that for subtract for inserting a gapm it is only minus 1, but for mismatch it is minus 10. So, you are more willing to tolerate insertion of gaps than mismatches of characters. So, that really depends upon what is it or what purpose are you defining the similarity function.

(Refer Slide Time: 15:09)



	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

As I said they could be combination specific. So, here is an example of a fine grained similarity function. This matrix 4 by 4 matrix gives you the similarities between any combination of the letters AGCT. You can see that the highest weight has been given of 10 to aligning A with A.

So, obviously, if you if you are going to have a similarity function which does that, it might even misalign some other characters if there are many As that would get aligned in the process, and that would be because your similarity function says that its most important to align an A with an A.

On the other hand, if you align T with a C, this is saying that it is neither good nor bad you are adding a 0 value to them. So, perhaps you have willing to wait all tolerated. But remember

that for all the four cases of aligning a character with itself, the costs are all positive. So, it is 10 for A, 7 for G, 9 for C, and 8 for T.

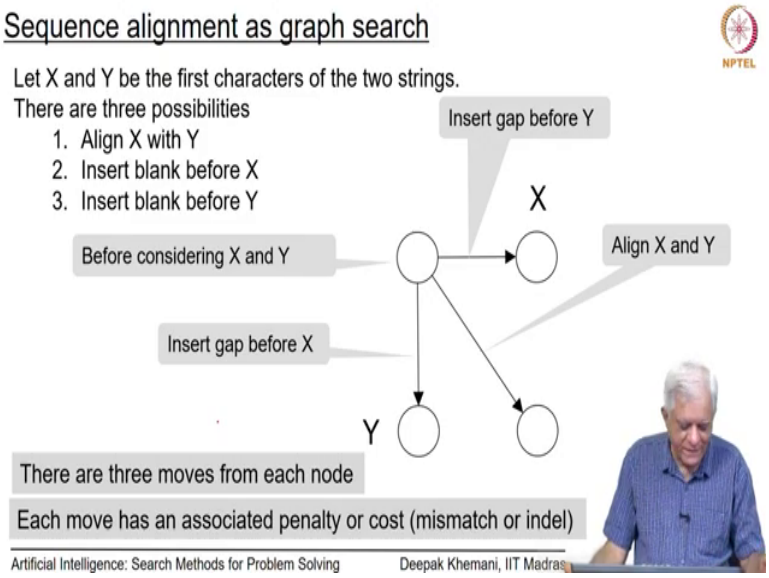
So, you would like them to be aligned with the same character. But if they are not aligned, then you have a certain cost to pay. For the case of T and C, the cost we have said is minimal or the penalty is minimal. But as you can see for the other misalignment, so for example, if you align C with G, then you give a negative cost of minus 5 essentially.

So, given this kind of a should I say complex similarity function if you take an example like this that you have these two strings and this is the alignment that is given to us. And if we assume that the insert gap insertion cost is minus 5, then the cost of this alignment would be as you can see the cost of aligning the characters A with C, then G with G, then A with A and so on and then we have this gap insertions, and then we have further alignments that will happen under this thing ok.

So, the way this has been written, you may not observe that this G is actually aligning with this G. T is aligning with T, A with C that is the alignment that is happening. And 3 gaps have been inserted which lead to a cost of 3 into 5 subtracted from the similarity values. And the rest of the values are as given. And you can see that for this particular alignment, the cost is 1.

So, obviously, you can try out different things and see what works best. And what Needleman invents did was that they posed this as a dynamic programming problem and they showed that by solving the problem, you can get the optimal alignment. And what we will do is now say that fine this is the dynamic programming problem, but we can pose it as a search problem in a state space, and then you can use heuristics search methods which will hopefully do things faster.

(Refer Slide Time: 18:25)



So, how do we pose this sequence alignment as graph search, so that is the core of this whole idea? So, let us say that you are taking two strings S 1 and S 2 and you want to align them. And start we start looking at the first two characters of the two strings and let them be called X and Y they could be any one of these C, A, G, T characters.

So, let X and Y be the first characters of the two strings. And this state this circle that we have drawn represents the state when you are about to start the alignment process. And what you are looking at are these two characters X and Y.

And you want to kind of model the problem to capture the fact that you can do one of these three possible things. So, there are these three possibilities that either you can align them align X plus Y, or you can insert a gap before X, or you can insert a gap before Y.

How can we see this in terms of the search graphs that we are generating? We start by looking at the simplest case when we align X with Y. This can be represented by saying that you are making a move, and you are going to a new case where X and Y are aligned.

So, if you were to imagine that this was all happening on a on an array as we will see shortly, then you can see that the new state is in the same column as X and in the same row as Y. Our state space would be like an array as we will see shortly.

So, but you move to a state where you have kind of taken care of both X and Y, and you have taken care of them by aligning X to Y, what is the cost of this edge, the cost of this edge will depend upon the similarity function. As we saw in the previous case, if you have fine grained similarity function, then it would depend upon what X and Y are.

In general, we can say that if X is equal to Y, then the cost would be 0 or something. But if X is different from Y, then we can have a uniform negative cost or uniform positive cost or a negative weight in terms of similarity which is the same thing. But the costs will depend on what is X and what is Y essentially.

The second possibility we talked about was insert a blank before X. And we can see that as the vertical move here that we are not traversing the string X, but we are traversing the string Y. So, Y you are at the place where you have already in some sense taken care of Y, but X is still being left hanging because Y is now going to be matched with the blank in the in the upper string. So, let us say the upper string is on the top which is horizontal, and the bottom string will go vertically on the left here.

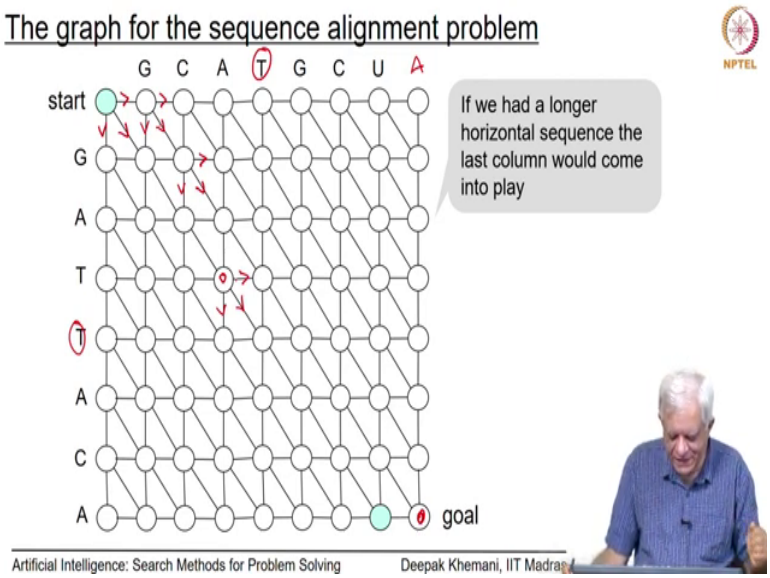
The third move is similar in nature which says that insert a blank before Y, and that means, that you move horizontally in this array like graph that we are about to draw. It means that you have taken care of X, but Y has still to be taken care of. And instead of Y, we had inserted a blank and that is what was aligned with this node X.

So, you can see that essentially there are only these three things that you can do with every character that you encounter as you process the two strings from left to right. The first character is X in the first string, and Y in the second string.

So, for each node in the graph that we are generating, there are three moves possible, either you insert a gap in the top string or in the bottom string, or you align the two characters together. And each move has an associated penalty or cost which is due to the mismatch or due to the insertion of the gap or not essentially.

So, after having done the first move, you would end up in one of these three successor nodes. And from there you would face a similar problem as to what should you do next as to what is the other characters that you are looking at in the string.

(Refer Slide Time: 22:47)



So, if you have let us say two strings in this example, the two strings are of equal length here. The one string is G C A T G C which is shown on the top. And the second string is G A T T A C A which is the which is shown on the left. And remember that the moves that you can make are either to the right or diagonally bottom towards the right or to the bottom.

So, you can move from here, you can move from here, you can move from here, from here you can move to this or to this or to this and so on and so forth. From here, you can move here, or here, or here.

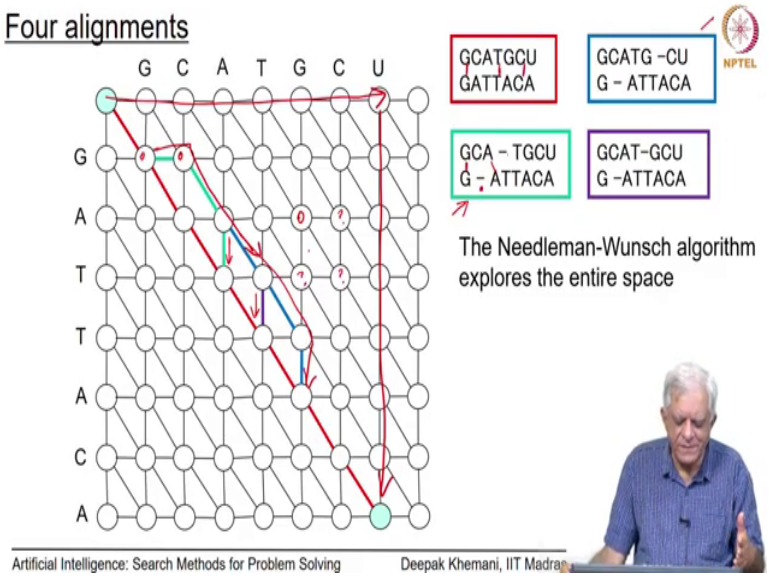
So, it is a directed graph we start at the start node on the top left, and we end up on the shaded node on the top in the bottom right because these two strings are of length 7 each. And so the number of nodes in this graph is 8 – 1 plus 8, because 1 we need for the starting position. So, it is an 8 by 8 array of nodes here, where you can move either to the right or to the bottom, or to the bottom right diagonally.

If we had a longer string here, for example, if we had a string let us say character A here then our goal would shift to this one here, and then it would be a non-square array if you want to think of it as an array essentially.

So, the task is now to find a least cost path from the start node to the goal node, and the edge costs are defined by the two strings that you are talking about, or the two characters in the two strings that you are talking about, because at each stage for each node you are so supposing you are here.

Then the question you are asking is should I match this T with this T, and then I would move in this direction, or otherwise if I were to insert the blank I would move either here or here? So, at each stage you are canceling two characters and making one of the two decisions, and eventually you will come down to the goal essentially.

(Refer Slide Time: 25:16)



So, for these two given strings here are four possible alignments shown in 4 different colors. So, the first one is red which is simply saying that there are no gaps inserted. And as you can see the paths that this alignment represents is the diagonal from the start node to the goal node without any deviations in any place essentially.

The second one is the one in the green which is this one here. And you can see that you have inserted two gaps. So, after matching G with G which takes you here, you are inserting a gap in the bottom string which is taking you here essentially, so that corresponds to the gap that we have inserted here.

Then you are aligning the next character A with A. Then inserting a back gap in the top string which amounts of this vertical move, and then the rest of it is the diagonal path in some sense you join the diagonal path. There are two more possibilities that we have depicted here. And

these are the four possible good alignments that you can talk about in this given problem essentially.

Of course, you can have a very poor case of alignment. You could simply go down this path, and then you can go down this path which would amount to saying that you first insert gaps in the bottom string until you have exhausted that, then you insert gaps in the top string, and that would be a path like this.

But obviously, you can see that the gap insertion cost might be too high. Whereas, in many of these problems that many of these alignments that we are seeing, you can see for example, that you are aligning C with C, you are aligning T with T, and G with G, and you are doing something which is reasonably good essentially anyway.

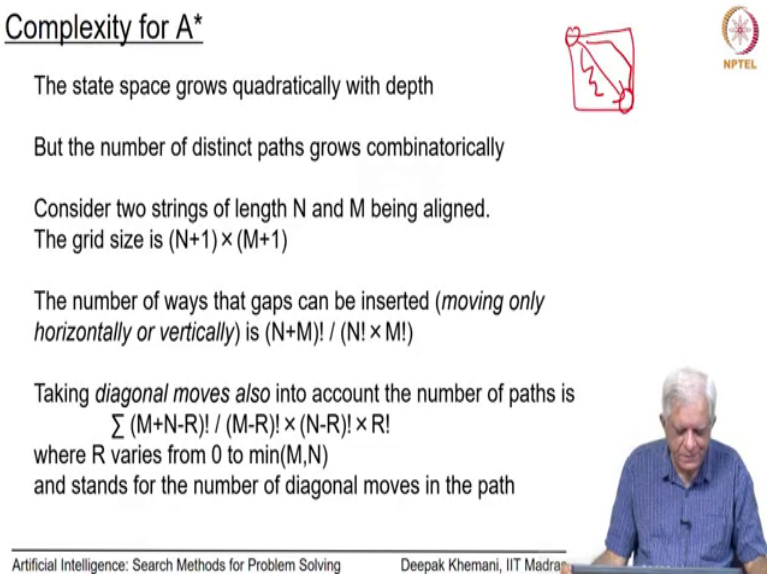
So, the other two good alignments are the one which is shown in blue here where also you have inserted two gaps, but in two different place and that is depicted by this path this deviation. So, you, you take off from there and then you come here and then you again meet the diagonal in some place.

And the fourth one is shown in purple where you rejoin the diagonal at this stage here. And you can see that these are the four possible different paths essentially. But anything is possible right remember that you can go any number of steps to the right, and any number of steps to the bottom, and so on.

What the Needleman Wunsch algorithm did like the Dijkstra's algorithm in some sense, it took the full graph and it computed for each choice point. That if you are here should you go to this node, or to this node or to this node and it computed the costs for each choice point. Look that all the costs and then trace back the path from the goal node to the start node, but it did not give you the optimal path essentially.

But now we want to kind of exploit heuristic search, we want to start searching from the start node and head towards the goal node. And we want to see if we can take advantage of a heuristic function in some way essentially.

(Refer Slide Time: 28:43)



So, let us look at the complexity from the perspective of A star here. The state space as you can see is growing as a quadratic with depth essentially. But as you go let us say N steps diagonally, then the space that you have in some sense covered is N by N. So, you have the matrix that you have left behind is N by N the space, and you could have come through any one of those path.

So, you come from here to here and this is N steps, then this entire array you could have come by this path, you can come by this path, whatever, you could have come by many paths, and

therefore, the space close actually only as a quadratic as you go away from depth. So, quadratic in terms of depth essentially.

But the number of distinct paths grows combinatorially that is the problem here is that there are a combinatorially there is a combinatorial explosion of the number of possibilities that you that you can explore in terms of the paths which are going from the top left to the bottom right. Even when the condition is that you can only move to the right or to the bottom, or you can move diagonally to the this thing.

So, if you have a, if you have a two strings of length N and M being aligned, and if we for the moment ignore the diagonals, and we say that we will only either move right or bottom as you would do for example in the Manhattan city.

So, either moving only vertically or horizontally, you can see that the number of paths – number of distinct paths is N plus M factorial divided by N factorial multiplied by M factorial. And you should work out these values for some small numbers N and M, and see that indeed the number of paths grows very rapidly indeed.

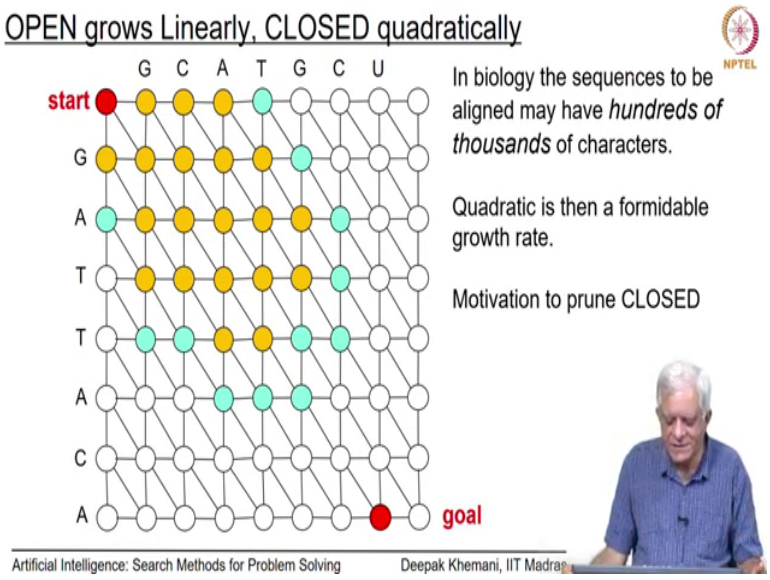
When you take the diagonals into account, this computation is a little bit less straightforward. The above problem where you are moving in a Manhattan like distance city is easier to solve and it has been studied by many people who study speak maths.

This second problem where diagonal moves are allowed is a little bit harder to compute. I will only give you the result here. The result is a summation to start with. It is not a closed form formula in that sense.

It is a summation of this figure which is M plus N minus R factorial divided by M minus R factorial multiplied by N minus R factorial multiplied by R factorial. And you vary R from 0 to the smaller of M and N the minimum of M and N. And if you try to work out this problem, you will see that this R corresponds to the number of diagonal moves that you are including in your path essentially.

It can be zero diagonal moves if you are moving only vertically or horizontally, but it could also be a complete diagonal move as we saw in the alignment that one of the alignments that we saw for the previous problem. So, it is a summation of all those things, and it is can be quite a large number to deal with.

(Refer Slide Time: 32:10)



Now, we have already observed that the state space grows quadratically. And this is just a snapshot of what could possibly happen with some heuristic function and some cost values which we have not depicted.

But in general we would expect that the number of nodes in open will grow linearly. So, as you can see at a distance d away from the top it can only be the sum of the horizontal and the

vertical distances which is essentially linear in nature. Whereas, closed does grow quadratically essentially.

And is that a problem? Obviously, with small graphs like the kind that we have drawn here it is not really a problem. But if you are working with bioinformatics or aligning sequences which come from biology, then each sequence may have hundreds of thousands of characters.

Now, just imagine that if you have 10 raise to 7 characters, well, that is 10 million let us say if you have 10 raise to 6 characters, then you can see that the square of that will take you to 10 raise to 8 size closed list. And these kind of numbers 10 raise to 8 is 100 million. Then if you want to show 100 million nodes, then maybe you do not have enough memory.

And if you can somehow work out an algorithm which will not keep the closed list as we have been talking in the last few sessions. Then we can in some sense overcome this quadratic growth which is tending to be formidable even in this problem. Remember that the problem the state space grows quadratically, but the number of combinations is growing like combinatorially, but the closed this is also growing quadratically.

And if the input strings are very long, then if you can cut down on this quadratic closed list then we can handle even larger strings. So, that is going to be the motivation for us to look at admissible variations of A star where we can cut down on the closed list. So, this is just a in some sense an oddity in the set of problems that we have been talking about.

In general when we talk about search spaces and we assume a constant branching factor for example, we have observed that it is open which closed much faster than closed. Because in an exponentially growing tree, open is more than all the closed nodes put together essentially.

As we saw in the beginning of the course and we are talking about that first search and breadth first search, we saw that breadth first search the open is exponential and it in fact most amount of work is done in the last row as it was said, and which is one of the reasons

we say that if we can move from that representation to a linear space representation like we did for DFID.

Then the amount of space that we will need would be considerably reduced essentially. So, that was the general case. In the general case open grows much faster than closed, but in this particular kind of problems which are combinatorial in nature essentially it is close which is growing faster than open. And we are kind of illustrated this by saying that this is what this space looks like.

So, we will begin in the next session by looking at algorithms which try to prune the closed list. And then we will move on to the more general case, and see how can we save on the open list essentially.

And then finally, before we are done with A star, we will have a look at an algorithm which tries to prune both open and closed which means that there is a possibility of tackling huge problems or problems without restriction on the size, but we will also see that it comes at a price.

So, just like DFID and ID a star came with a price these algorithms also too will come with a price, and the price would be much lower than that was in the case of id A star. So, we will take that up in the next session.