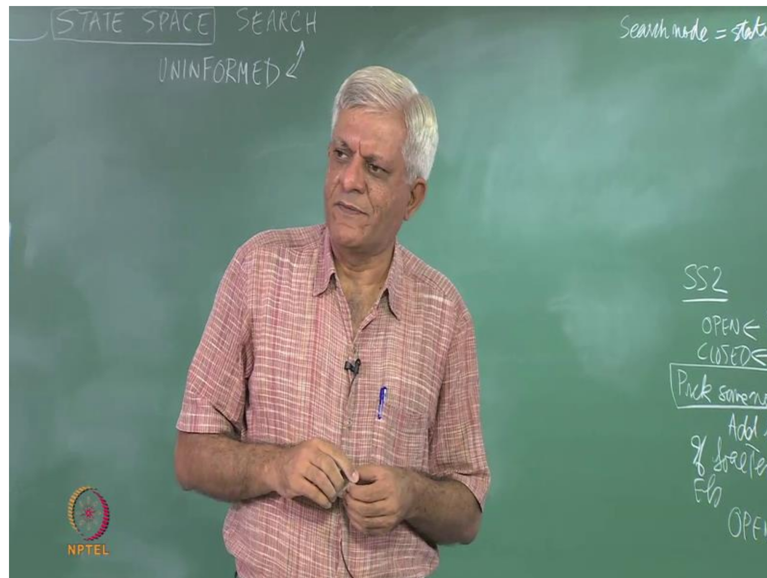
Artificial Intelligence
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

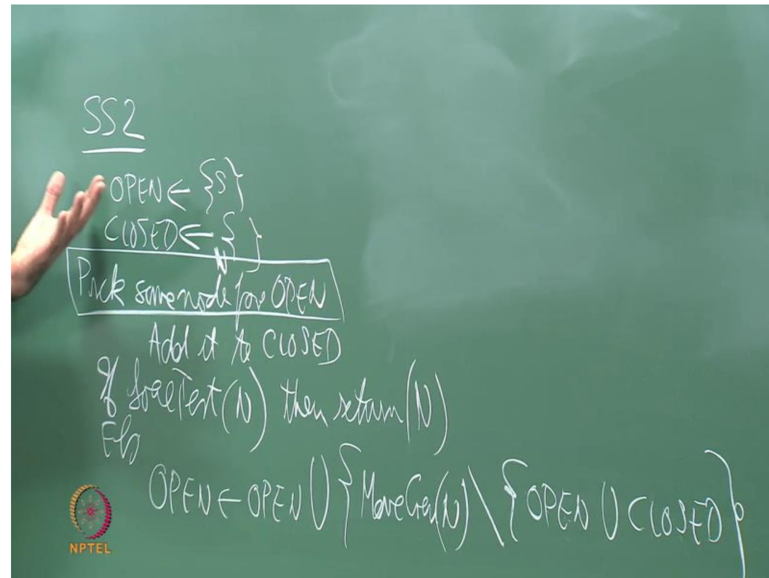
Lecture - 07
Search - DFS and BFS

(Refer Slide Time: 00:23)



So let us get back to state space search and valuation that we are looking at, we are call uninformed. Valuations which we are looking at today, I mean from in the sense, they do not exploit any knowledge of any kind from the domain. On the next class when you meet on a Wednesday, we will try to see how to get around this, how to exploit some knowledge from the domain essentially.

(Refer Slide Time: 00:53)



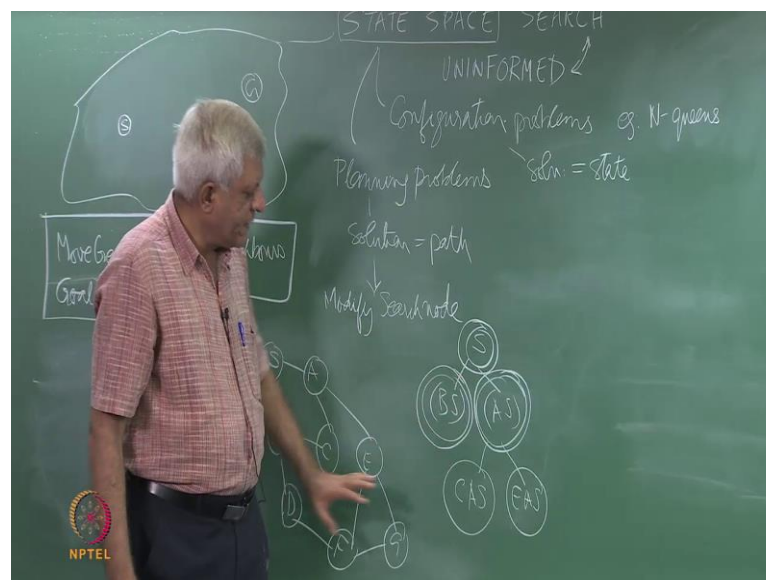
So, the last algorithm that we saw was this, simple search to in which we had two sets open and close, open contains a set of candidates set we want to inspect, and closed contains, the set of candidate that we have already inspected. And the idea is to keep generating new candidates, and adding them to open, and how do we can generate new candidates by, removing from move the output of more than anything which is already on open or which is in closed essentially. So, we will only get new candidates, and in this ways, we will only had new candidate to inspect.

And if the state species finite, you can see that eventually, whatever states are reachable from the start state, they will be explode at some point or the other, and it will come out of this loop only, it open becomes empty. So, there should be a check here, if open not equal to empty then do this, open him to empty report failure essentially. But, let us assume that there is a solution to the problem, which means the goal said that we are talking about, is reachable from the starts state, in which case we would find the solution that some point or the other.

So, let us take the main missionaries in animals problem or the man, goat and lion problem that we have been talking about. It is possible for the man to take the goat, and lion and the cabbage on the other side, what will our algorithm do, this algorithm that we

have said, what are the incase dent and what is the algorithm giving us. So, what are the interested in it start with that, what are the, what should the man do right, man should first take the goat, then keep the goat on that side, comeback with the empty boat, then take the lion and all these kind of step, what is the algorithm giving us? It is only giving us to state end with satisfy the goal test that does not help us in any way, to solve the problem.

(Refer Slide Time: 03:11)



So, that is the second problem with this solution, and we have to address that, but before we do that, let me also clarify the there are two kinds of problems, one is call the configuration problem, and the other kind is planning problems. In planning problems, the solution is a path; in configuration problems, solution is a state. So, we can actually distinguish between these two different kinds of problems. So, the river crossing problem is the planning problem, we want to know, what are the sequence of moves the man must do, to solve the problem.

But, there are problems which are configuration kind of problems, so for example, the N queens, so you must be familiar with the N queens problem, I presume. The task is that given a end by end chess board, you have to place N queens on the board, in such a manner that no queen attacks, any other queens essentially. So, that is a well known, very

commonly studied N queens problem, there are variations with this, which say for example, every queen must say attack exactly to other queens. So, that is another configuration problem.

So, in such a problem, the N queens problems for example, the solution is only the final state that we ((Refer Time: 04:41)) if you can show that such a state exists, then we have solve the problem, it does not matter they. In fact, there is no notion of a path there, I mean unless, you say this is the first queen to plays on the board, and this is second queen to plays on the board, that does not make sense. We are interested, then some configuration and such problems are called configuration problem, for which this algorithm is fine because, it will return to us a state with satisfies the goal constraints.

Whereas, for such problems, we does not work, because is only telling us what is the final status, how do we solve this problem. So, for planning problems, how do we have to modify the search node? So, there we said the search node was the state essentially. For planning problems, we have to modify the search node. So, what do I mean by the search node, the node generated by this algorithm, what we put into open, what we take out from open, and you know things like that essentially. And we have to modify it in such a way, that it contains the paths information essentially.

Now, one simple way of doing that, is to store the entire path as the search nodes essentially. So, let me use a small example, supposing this is the start state, and this is the successors state. So, let us a some small states piece we have, some random graph I drawn, this is the start state, and let say this is the goal state, and we have to find the path from the start state to the goal state essentially. So, you say, so you will go, you will a algorithm some will begin with start, then the move gen function will return it is neighbors, which is A and B.

And then the algorithm will inspect a whether a is a goal or not and that kind of things, so it will do all that stuffs. But, I want to know the path that the algorithm finds, from start state to goal state. So, I modify this search node, to store the entire path. So, what does that mean, I start with this node itself then my successors; I call them as B S and A S. So, I keep a list of nodes as a, so a search node is a list of states or list of state nodes.

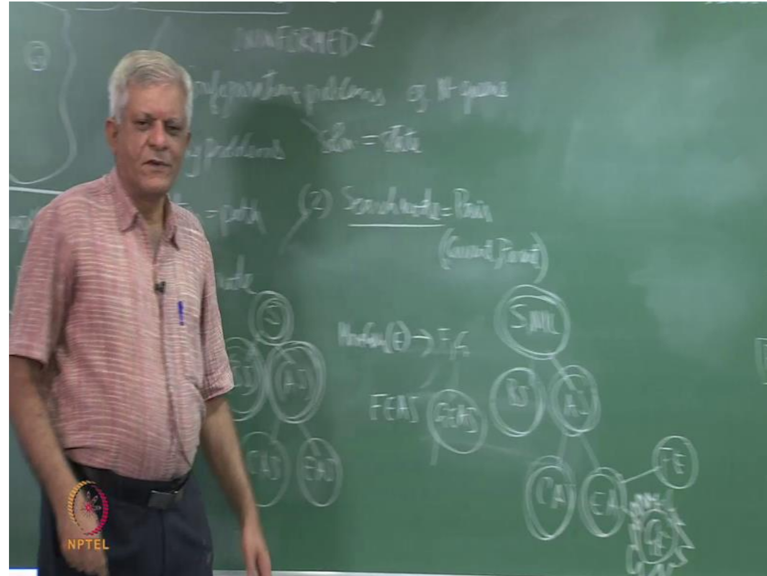
So, this list is the list of two nodes B S essentially, with basically says, I came to B from S, this says that I came to A from S essentially.

Then if I about to expand A, then I could the let us assume that we are not going to add S to successors of A, only C and E. So, I will write that as C A S and E A S. So, this is one approach, that I am modify I am search node, to store the entire path. Of course, now I have to modify my algorithm little bit, which I am not doing, but I am leaving does not exercise for you to do, when you pick a node from the search node. So, this is, so these are the search node, so this is the original node.

And then, in the new scheme of things, double circle stands for node gen closed, instead of deleting nodes, because they not deleting nodes anymore, pulling them in closed. So, we put this in closed, I do not know I hope it not to confusing, put this in closed and so on essentially. So, these are the two nodes are in open and this three, let say we have inspected already, when I pick a node from.

So, this is I mean, this is my search node, it is a list of states, I have to extract the first element from this, which is C or E depending on which node I pick. So, I extract seat, apply the goal test to C, same goal test function I can apply to C, and if it is fails. So, let say C any C does not have any children, so let say I pick E, I apply the goal test to E, and goal test fails, so what do I do I generate the successors of E, again the same move gen function, I can use to generate successors of E, but I must append that.

(Refer Slide Time: 09:22)



So, what are the successors of E returns to me in this graph F and G, so I must take this F, I must take this G, and append and put it at the head of this list. So, I must generate F E A S and G E A S. So, I have to modified the algorithm is little bit, but notice that I can still use the same move gen function, and I am only going to applying to a state. And I can use the same goal test function, because I will extract the first state from this, and apply the goal test to that, and then apply the move gen function that is essentially.

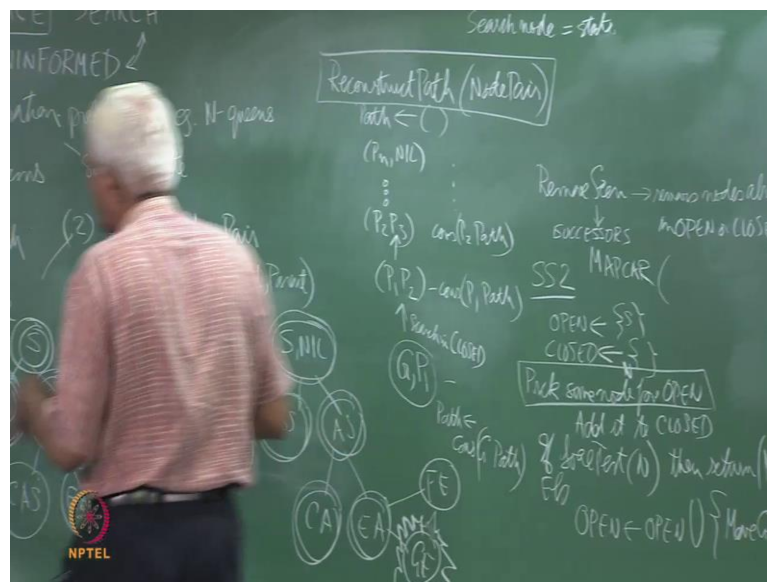
So, I will leave this as a small exercise for you to do, we will inset use another approach. So, this is one approach, to convert everything into a path, the second approach is to only store the parents essentially. So, search node is the pair, we will call it a node pair, and the pair is current and parent, and let say it is a list of two nodes, in which we store the current node. So, we will always be interested in the current node, we will apply the goal test to current node, will apply the move gen function to current node and so on essentially.

But, we will only stores, so here we are storing the entire path, up to the start node from that, which of course, makes the task of returning the path must simpler, I have to just take this and reverse it and return it as output S. So, for example, when I generate this, and this is the, this the satisfy the goal test function, then I have to just take this and say

the path is go from S to A, A to E and E to G and that is solution. So, I have just reversed this list, and return it as a path. Now I am saying, that instead of stoling this entire path as a search node, uniformly store a search node as a pair, made up of the current node and the parent node.

And my route will look like S comma nil, a nil stands for no parent, and then this will be again B S, this would be A S as before, but this would be now only C A and E A, and when I generate successors of A, I will get F E and G E. So, all this is gone into closed, and let say whether does not matter, whether we have seen this or not, and when we pick up this node. So, let me say this stands for the fact that we have found the goal node, we have to still return the path, but now you have to do a little bit of extra work, to reconstruct the path.

(Refer Slide Time: 12:46)



So, we need than algorithm, which you will call as reconstruct path, and to this algorithm we will give a node pair as input, and which node pair the one whose first elements satisfy the goal test. So, this for example, G E will give this as input to G E, so what do we want to G do with the G E. So in general, I will have a goal node and parent node, I will call it P 1. So, this is I am describing this algorithm reconstruct path algorithm. So, the input to that is a goal node and it is parent node. So, it is a node pair and the

particular node pair is a goal node and it is parent node. And so I start constructing the path, I say I had.

So, initially as a path gets an empty list, and then at this point as a path get cons G path. So, I am using the list cons functions, because it is a convenient mechanism to add something to the head of a list, so I have a list called path, and I am adding this node G at the head of the path, then what do I do, how do I go to the parent, parents parent, I want to go to the grandparent ((Refer Time: 14:20)) parent. Of course, I can find from here, but from the parent, I will have go to the grandparent, they will a find the grandparent corresponding.

Student :((Refer Time: 14:34))

But where do I find the corresponding parent.

Student: if we store the parent ((Refer Time: 14:39))

But, how do I I am looking for a node pair of the kind P 1 P 2 correct. So, I know that P 1, I came to G from P 1, I am trying to find out, where did I come to P 1 from, and that that is call it P 2, where will I find this P 1 P 2.

Student :((Refer Time: 15:01))

I will find it in the closed list why, because I am putting all these things into closed. So, for example, G E is what I have, and I am looking for something like E followed by something, which is E A in this case, and will find it in the clues. And likewise the parent of A, I will find in the clues and so on. So, search in closed, so I search in the closed to find this node pair, everything is a node pair, opened with consists of node pairs, and search consists of node pairs, and close consists of node pairs essentially. I find this, then again like this, I find P 2 P 3, and I keep doing that, till when will I stop doing this search.

Student: ((Refer Time: 16:00))

Till I find the route node which is, characterized by some P n, followed by nil, and at each stage, I will do this, cons P 1 path; here cons P 2 path and so on and so forth, I will keep doing that by the time I am come here, and found this, come out of this loop. So, all this is happening inside one loop, going from parent to grandparent, till I find that there are no more grandparents, at which point, my path for the win constructed, because I would cons this P n to path. So, I would know that, I start with P n go to P n minus 1and so on and so forth.

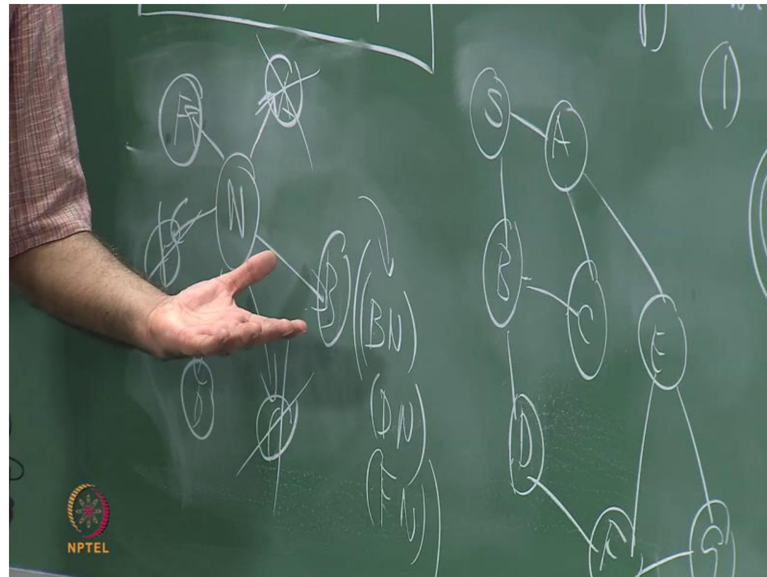
So, I need this extra bit of work to be done, to return the path, but what do I get in that advantage is that I have a uniform representation that everything is a node pair, open content consists of node pairs, close consists of node pairs and so on and so forth. So, this part for example, searching in the closed, I will leave it as a small exercise to for you to work on. And write this whole algorithm in fact, for reconstruct path. So, we have this algorithm call reconstruct, a supporting call reconstruct path, we have also this supporting in that is call it, remove seen, which removes nodes already in open or closed.

So, let us say that we have now move completely to a list based representation, where our path is a list, node pair is a list, everything is a list essentially, open is a list, close is also a list. So remove seen, when we are generating the new nodes for open, the remove seen function should remove things which are either in the open list or in the closed list. Now, we must be careful here, our open and closed has been modify the little bit, they have become pairs. So, you have to take care of the fact that you are looking at the correct element there, so I will lead that exercise for you to do.

And, we will assume that, we have something like a, suppose those of you who are studied list or something similar, something like a map car function, which does the following that. So, this removes in will represent, will give as a list of successors, so we will apply the move gen function, which will gave us all the neighbors, there removes in will remove from those neighbors things, which are already in closed or which are already in open in it will give us new list. From this new list, we will apply this map car function to, say that take each of the successors, and construct node pairs with the parent

essentially, so whatever it doing.

(Refer Slide Time: 19:42)



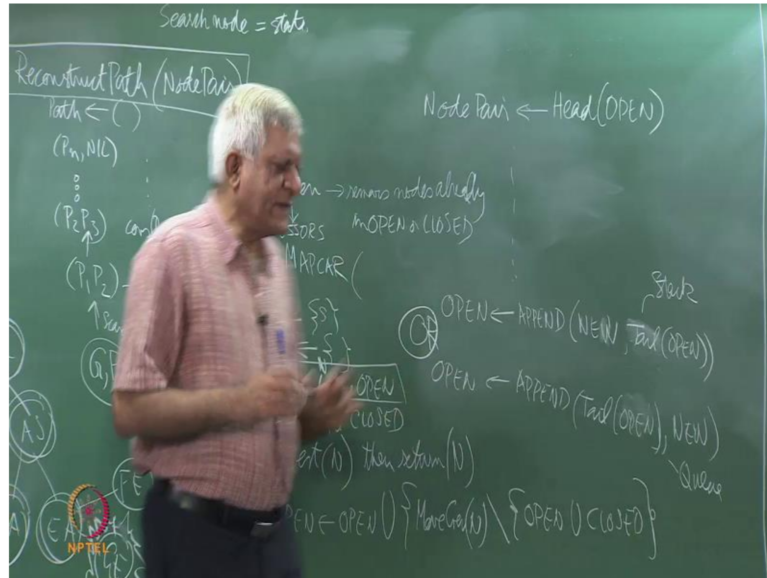
Let me draw it here, so there is some node N, we will call move gen, it will give us some successors. Let us call them, let say A, B, C, D, E, F then remove seen will remove some of them, let say this one and this one. So, we are left with B D and F; then I want to call some function, which will essentially give me this list of B N, D N, F N. So, it should give me a list of three elements, in this example, of the three successors, already converted into node pair form. So, I can just take this and upend it too, open list essentially.

So, my algorithm is still very similar, open is a list, I extract some element, from the list, which is a node pair, from the node pair I extract the first element, which is the current node I am interested. Apply the goal test to that, if it works or if it returns to then I call the reconstruct path function, with that node pair and it will give me the path. If he does not work, I generate successors of that element N, node N to move anything that I have seen or I have put on open already.

And construct node pairs from so the successors of N or B D and F, so I want B N, D N and F N. So, these are the B D and F are the successors, but I want to construct the I will

want to remember that N is a parental B; N is a parental D; N is a parental F. And I will add this to open, and the same cycle will continue. So, that is a algorithm that we have written, so the only thing that is remains is, when he said pick some node from open, let us pin that down as well.

(Refer Slide Time: 22:06)



And we will do the following node pair, I call it node pair as a variable name, I will always pick the first node from open, open is the lists now, and the simplest thing to do with the list is to remove the head element. So, I will just say I will remove the head element. And so, what will this give it, this give me a node pair, I will extract the first element from that, and do all this goal tests and everything is to that. So, one thing I have pin down is, that we will always take the first element from the open list.

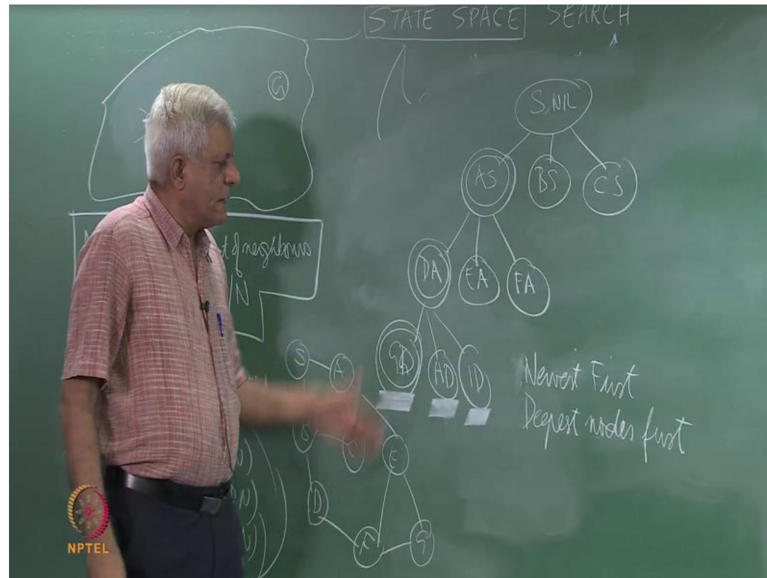
Open is now a list essentially, what should be the first element of the list that is the next question, that will really now determine the behavior of our search algorithm. So, there are two possibilities, so somewhere down this line, I will have open as the following thing append. So, I want to append two lists, what are the two lists, one is the old open minus the note pad that we have removed. And the second is, the new elements that we have generated by this process, this list that we have, I have to append this two lists to found the new open.

Again I can append them in one order or the other order, and that is really going to this side matter. So, let us choose one order first, which says that append new. So, this list, number this is the list of node pairs, I am calling new, and this I went to a generate this new, when I have inspecting the state, found that it is not the goal state generated it is successors calling the move gen function, moved duplicates from there, made pairs and call this a new, so this is new. So, it is in the form which can be added to open, so that is new, and I will just use rest, all you can use, tail assuming a not removed in that step from open, because this only which return the head, here I will take the tail of open.

So now, I have a completely deterministic algorithm, there is no ((Refer Time: 24:54)) statements about pick some node or add this node to open and so on. We have specified everything completely; we have said that the new nodes will come at the head of the open list, which means they will be inspected first. And I will always pick the head of the open list. So, we want to now, look at these two options that we had, what is the other option? Other option was the opposite, was to change the order. So, I have to choose between this, either this or this, which option is better. Now, what do we get from what rather how to be analyses choice, so this is the choice that we have to make in our algorithm, should be up should be put new at the head of the list.

And keep the old elements behind or should we keep the whole elements first, and put this in the ((Refer Time: 26:12)) So, you would have recognize this two data structures, that we have kind of simulating here, this is the queue and this is a stake, this like a stake. So, we can maintain open is there as a queue or as a stake, what is the repucation of these two choices, is what we want to, inspect next. So, let us take this option first, where we are looking at the new nodes first, so newest nodes first, what does that meaning in terms of the search trees that we were talking about.

(Refer Slide Time: 27:30)



Let us draw the search tree. So, when I draw the search tree, I will not draw the node pairs and everything, all let us any you can draw it does not matter. So, let say S nil and A S, B S, C S in some graph now this is an ordered, this I am depending on the order and which may move gen function gives me this A B and C, they will go in some particular order that is not worry about that. So, I always choose the left most first, so I think this and add let us call it D A, so the next one will be this one and so on.

You can see what is happening, this algorithm in which open is treated like a stake, as this greedy like behavior, in the sense that, the latest nodes that have been generated are explode first. So, these are old, so these are the latest, out of them one we have chosen randomly, and officially without loss of generality we always chosen the left one, it does not matter really. Because they were generated where move gen function, and we have not specified them order there essentially. So, we officially a chosen the left most, these are the newest; these are little older; these are the oldest.

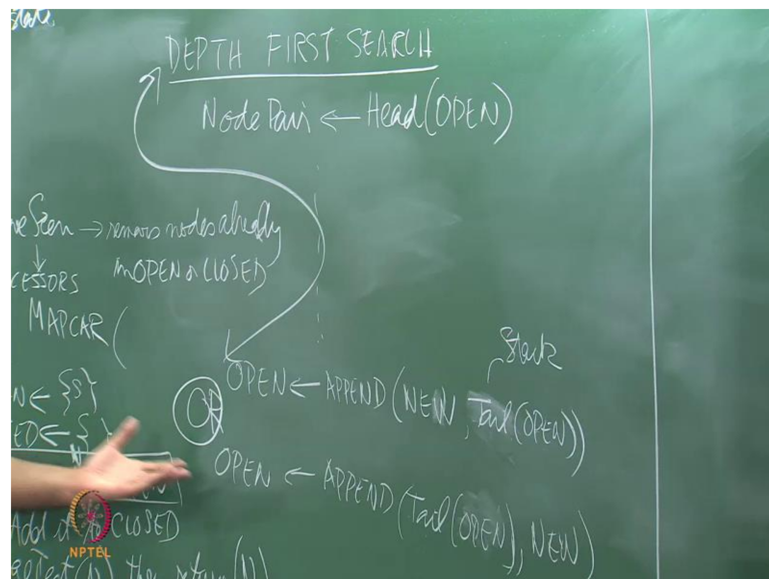
The algorithm always picks the newest node first. In terms of the search space the states space, what is this algorithm doing? It will take go from the start node, it will generate some successors, it will choose one, it will generate the successors, it will go there and so on. It will dive into the states space, you know headlong without looking left or right.

Till of course, it heads a dead end, what do you mean by dead end here? When there are no new successors along this path essentially. In which case, for example here, if this was to be a dead end, that they were no successors of this G.

Then automatically this would be inspected next, because that is that would be next in line and open, if this was also a dead end, then automatically this would be inspecting next essentially. And if this for a dead end then automatically this would be ((Refer Time: 30:29)). So, the you can see the path going like this, and like this, and like this, and like this, it dives into the search piece, there is a danger of course, that if the state space is infinite. If I say you know, find me two numbers or three numbers, whose powered to three A is to three plus B is to three is equal to C is to three.

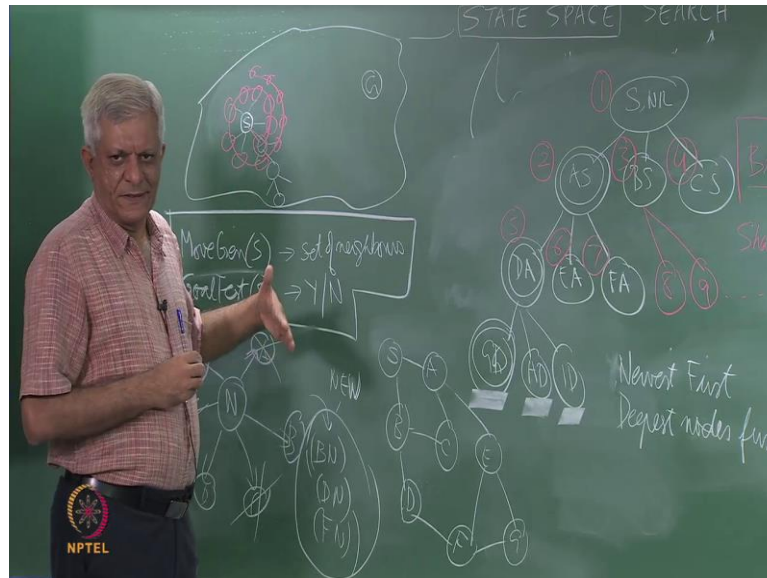
You will just try generating newer and newer combinations, and you will keep diving into thus search pairs. So, if the states, search space is infinite, there is the danger that this algorithm will get lost essentially. If we look at this search tree, we can also called it as the deepest node first, going down this direction of all the open list, who was the candidates in open all these single circle nodes, this is in open; this is in open; this is in open; this is in open, these two I will open. It always picks that deepest node first essentially.

(Refer Slide Time: 31:46)



So, you should not be surprised at this algorithm is called depth first search, this algorithm ((Refer Time: 31:58)) as oppose to that. The other choice we had, was to maintain open as a queue, and as you can imagine, what will happen with a queue is that.

(Refer Slide Time: 32:24)



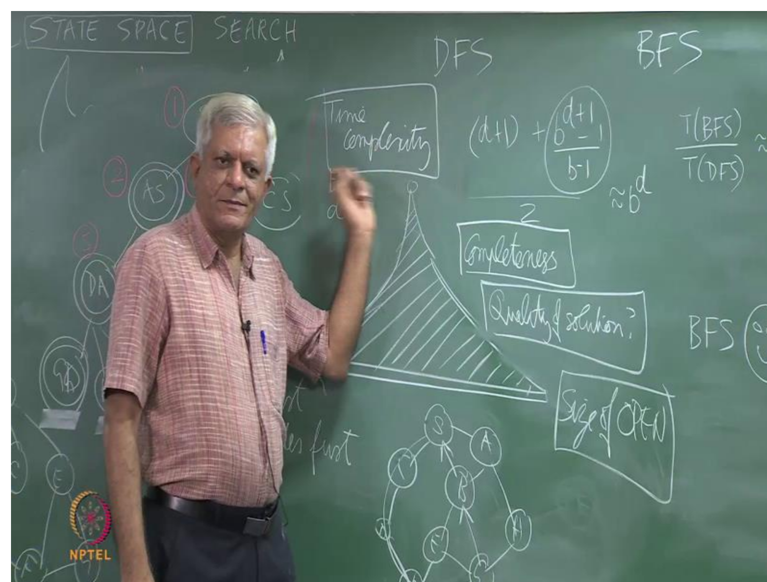
I will write the order here, this is one first node to be inspected; always the start node is the first node to be inspected. This will be the second one, they will be generated, but now it is a queue, which means these two will go behind these two. So, this should be the third node; this should be the forth node; then this would be the fifth node; and this would be the sixth node; then this would be the seventh node; the eighth from would be this child here; then nineth ((Refer Time: 32:54)) child and so on.

It has a exactly the opposite effect, depth first search dives into the state space, this algorithm chooses the shallowest nodes first. Even a choice of nodes, in this search tree, it always pick the one which is closest to the start node essentially. And you know the name for this algorithm, breadth first search, breadth first in the sense, first this layer completely; then the second layer completely; then the third layer completely and so on and so forth. In terms of the states space, what is this algorithm is doing, it will come here; then it will go here; then it will go here and here and here, in this particular order, it will clever the states space.

Now, notice that both these algorithms are totally oblivious of the goal state, goal state is somewhere here, they do not cares, only place where the goals set plays a role, is in a goal test function. If you have reach the goal state, we has the algorithm can consume whether you reached or not, but given the set of choices, it has no sense of direction that. I should go in this direction or I should choose this successors, that will come to, when you look at heuristic searches, but these two algorithms are blind. So, we call them blind search algorithms or uninformed search algorithms.

One of the ((Refer Time: 34:48)) to stick as close to the sources possible, the other has opposite tendency a going as far away as possible is essentially. So, let us do a comparison of these two algorithms, that depth first search and breadth first search. So, I can remove this, what are the parameters on which we should compare? We will look at four criteria, what is the most common criteria for comparing algorithms.

(Refer Slide Time: 35:45)



So, D F S verses B F S, the most commonly use criteria is time complexity. So, let us assume the simple search piece. So, let us take the N queens problem, in the N queens problem, let say that the way that you will proceed is, you will place the first queen, let say in the first row, then the second queen in the second row and so on, up to the n th queen essentially. So, the search space would be, end steps the solution will always be

end steps along.

And the branching factor is constant. So, if we have a search tree like this, why do I provide like this, because the number of nodes are going to increase exponentially. So, if I branching factor is five let us say, I will have 1 node at this first level; 5 nodes at the second level; 25 nodes at the third; they will for second level whatever you want to ((Refer Time: 37:14)) 125 in this level and so on. It is going to multiply by the factor of branching factor every time, and any and in such situations where you multiply at a every stage the think tense to go exponentially.

So, this is how search space nodes the search trees looks like, so let us forget about that five thing. So in general, we have given a branching factor of b, and let say the solution is a depth d. So, it is let us say is a d queens problem, you place d queens and then you are done essentially. Let say for argument sake that there is only one goal node, in practice of course, for example, to the N queens problem, there are many solutions. So, they would be many nodes which are the goal states, let us assume that there is only one goal node, in some problem which is similar to N queens, what would be the time complexity of depth first search.

So, before we do that, in the last layer, there are b is should be d nodes, and all the internal layers nodes, how many are inside, up to one layer less, up to one layer less, how many are the total internal nodes b is to d minus 1 these are the internal nodes. Now, the first into observe, there is if we ignore that minus 1, for large b and large d you can ignore that minus 1 on the top of site. So, you can say this is roughly b is to d divided by b minus 1, even this minus 1 you can ignore for large b, but let us we are not bother about that. The first thing to observe, is that is d th layer contains more nodes then all the previous layers combined.

So, this is the nature of exponential growth that every time you go one level, further away. The amount of work which do a that level, and by amount of work will mean inspecting that many number of nodes, is greater than all the work that I have done before that essentially, that is a feature we will use. I think it will have to be in the next class, but we will see that. Now, what is the time complexity of this, so will next some

simplifying assumptions that the goal not can be anywhere from here to here. So, in the case of depth first search, if it is here, it will just inspected after d plus 1 inspections.

So, let us say d plus 1, plus seeing the entire list, which is the entire tree, the ((Refer Time: 41:02)) it and let say divided by 2 search some ((Refer Time: 41:06)) proximation essentially, which if you work out, will turn out to be roughly b is to d essentially, of the order of b is to d, of b is to d by 2 essentially. For this one, either it has to inspect the entire sub tree, up to this point, see noticed that depth first search, were it finds a left most goal, it just only inspect these nodes, just goals on this path and finds a goal node.

So, it only inspects a d number of nodes on the way, but when breadth first search comes to this, it has to go through like this, and like this and plot through this whole thing one slowly, slowly. It has to inspect the entire sub internal sub tree, internal tree before it comes to this node, and for the right most of course, both of them I inspect the entire tree. But surprisingly, I will leave that for you to work out, so this is the internal tree plus the full tree and on the divided by two average, and then you can makes simplifying assumptions, assuming that b and d are large.

So, you can simplifying the expression, but the time complexity for B F S divided by the time complexity of D F S, after we make the simplifying assumptions is roughly b plus 1 divided by D. Death first search takes a little bit more time, but not significantly much more, you b is large for example, if b is 10, then it is 11 by ten times essentially. So, if not too much, it is kind of equals in some way, for both of them, as for as time complexity goes, what are the other factors we can compare them on, completeness and what do we mean by completeness, the question we asking is.

If there is the solution, which means, if there is the path from the start state to the goal state, thus algorithm always fine it, if it is, if he does than will see the algorithm is complete. Now, breath first search is; obviously, going to be complete, because as you can see, is going to go slowly in circles and if there is the path, knew little find it, whatever death first search, if the search face going to be infinite, which means this boundary was not there, and even number theoretic problems are an example of that, then it will just go of in some direction. And the goal may be here, it may just go of some

directions and the danger always exists essentially. So, completeness for death first searches, if the search pair is infinite, it is not completely you can get lost.

Breath first search will still find the solution, because it is not going to go off in any direction, it is going to just gradually expand it is, sort of set of nodes it has inspected, till it eventually it will hit this, and then it will find the solution. So, breath first search will become will work, even for infinite graph. For finite graphs, both are complete, because eventually both will inspect all part essentially. So, completeness, breath fast search is a slight advantage that if the graph is infinite, it will still weak complete essentially.

Quality of solution, the only ((Refer Time: 45:11)) we can talk of at this movement is the number of the length of the path or number of nodes in the solution path essentially. And we assumed that shorter paths are better, what do you think about these two algorithms? In terms of quality, is any of these algorithms guarantee to give you an optimal solution, you should speak a little bit louder.

Student: ((Refer Time: 45:58))

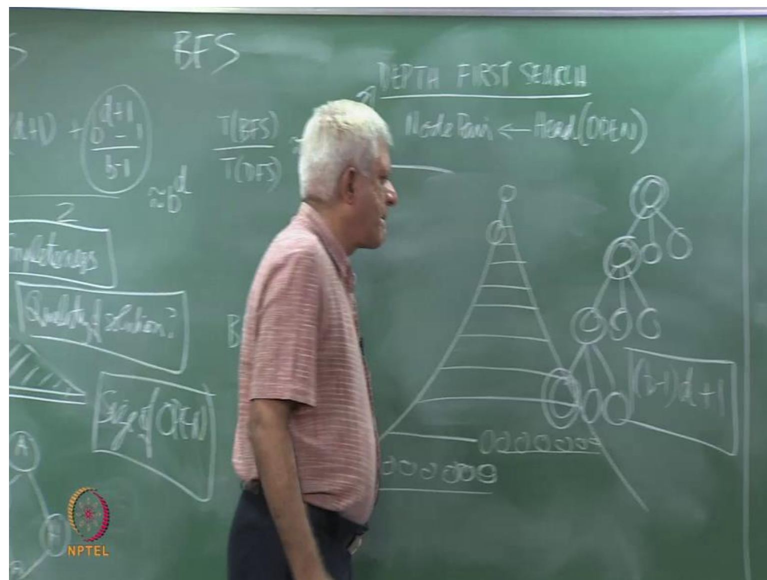
B F S now we have to say, what do you mean a might ((Refer Time: 46:04)) even B F S might give up to essentially. It depends on the order and which you done it moves essentially. Let us take a graph start A, B, C, D, let say we take a graph like this, you have to go from S to G, what will and let us assumed that, the particular order and we choose the left most on the board first, what will depth first search to, it will go from S to C, C to D, D to F, F to G. And let it will say I found the path, but the path I could have found this one, what we breath first search do, it will go from H to C; then it will go to B; then it will go to A; then it will go to D; then it will go to E; then it will go to G.

G would be generated as the child of B, and you should work this out as an example yourself. And the parent of G would be B, and the parent of B would be C, breath first search will always find the shortest path. So, I want to you to change this example and un simulated, to see that depth first search and breath first search. Depth first search always guarantees ((Refer Time: 47:26)) in terms of this you can see, there is gradually moving away from the source, and when it hits the goal, it will always find the shortest path to

the goal. But, we have to sort of reason, it out to a little bit, quality of solution B F S scores high heavily, it always current is a shortest path essentially.

Let us look at the last criteria, which is size of open, now we can see that, this time complexity the way we are measuring it is in some sense, measure of the size of closed, because the number of nodes that you have seen, will be the size of closed essentially. And we are assuming, for the timing that it takes you constant time to inspect get nodes out of close, and all that kinds of that. So, it is roughly call response the size of closed, what about the size of open, what is the length, what is the size of the open for the breath first search first. So, we have to visualize what the algorithm is doing.

(Refer Slide Time: 48:49)



This is my search space and for breath first search, what with the open list look like, open would, so you will inspect this; inspect this; inspect this and so on. So, let say this is, let say this is a closed list, we have inspected so far. So, open would be all these nodes, and all these nodes here, which are the children of these nodes essentially. This is the open would list look for, depth first search, and what is the size, how is it growing in terms of depth, it is growing, remember that, then b is to d nodes here and b is to d plus 1 if you want to say. So, it has got some from b is to d know and some from b is to d plus 1 row.

In general it is growing exponential with depth the deeper you go, the size of open explodes, it becomes exponentially ((Refer Time: 50:02)) what about depth first search, depth first search would come here. So, let us let we draw separate this thing here, let us say it is three children here, this is gone into closed; this is gone into closed and so on, what is happening with depth first search? As I go deeper into the search piece, it is going linearly, how is it going linearly, because as I go deeper, I am adding a constant number of nodes to open. I am adding basically two extra nodes, at this level they were 3 plus 1 I am going to inspect, so this thing then two extra here; two extra here; two extra here.

In breath first search, I am multiplying by a branching factor every time that is it is why growing exponentially. So, you should satisfy yourself that the size of open, for depth first search is b minus 1 into d plus 1, because that every level I am going to keep b minus 1 nodes into I am going to add b minus 1 nodes into open, one nodes I will inspect this thing. So, b minus 1 that d plus one extra node, because at the last layer, I will have one extra node essentially, so size of open D F S means hands down, grows only linearly as oppose to breadth first search for which open grows exponentially.

So, we have seen this two algorithms, for finite graphs, goals are complete, goals are roughly the same time complexity, but one of them wins on the quality of the solution, which may be important. If you have planning, let us say trips from here to the moon or to mars, then thus smaller number of trips is going to save your lot of money. Let us say from here to Bangalore the same thing, almost the same thing, quality of solution, breadth first search guarantee is you an optimal solution. And you must have survey this in other context, open requires only linear space for, I mean depth first search requires only linear space problem and that is a big, big plus.

Time complexity is bad, we will addressed that using heuristic methods, try to force a search to go towards the goal as we have seen. This one is the ((Refer Time: 52:45)) close to home, this one is going off in some random direction; know none of there is sort of looking towards the goals essentially. We will see that, then the complete heuristic search, but even a blind search algorithm or in from search, can I combine, can I device an algorithm, which will combined these two plus points of this, guarantee that it is an

optimal solution, yet require linear space for open.

So, I will leave this as a small thinking exercise for you, when we meet next on Wednesday, we will look at an algorithm, which does this. So, it should be nice, if one of you can think of it without reading it from somewhere else. After we discuss it here, you can go and read it out, but there is an nice algorithm, which will combine these two things, and it is a blind algorithm, so see you on Wednesday.