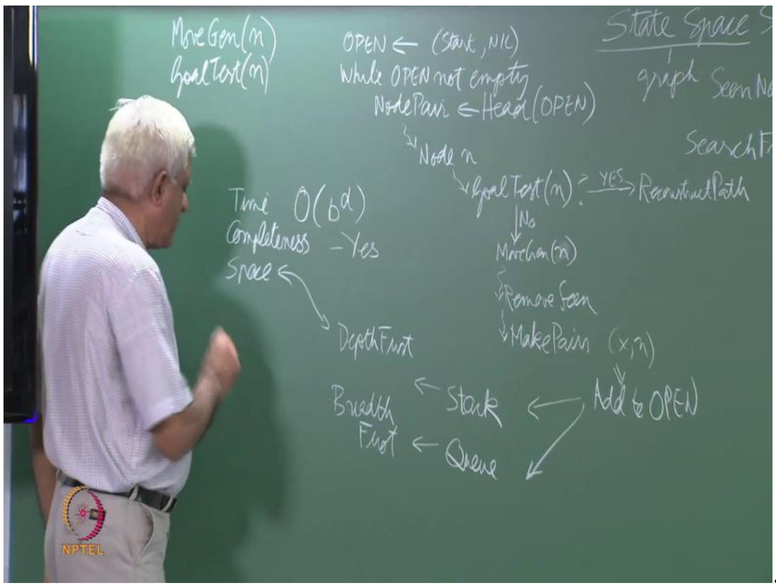
Artificial Intelligence
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 1
Lecture – 8

(Refer slide Time: 00:14)



.

Let us continue with state space search, let me just do a very quick recap, and then we will continue from there. So, the state space is a space in which made up of many states where each state represent the particular situation. And the states are connected together by the moves you can make of the decisions we can make. So, that if you are in a given state, and we had defined this function called move gen.

Which takes, n we will use for a node in the state space. So, the state space is implicitly a graph I say implicitly because, it is not available to us. It is not that the graph has been given to us. And we have to find a path in that graph we have given start state, and we are given either a goal state or the description of goal state. So, for example, in the river crossing problems we had given a goal state that everybody must be on the other side of the river. But on something like n queens here given a description of the goal state it will says that, place and queens that now, no queen attacks any other. Either way we have given some idea about the goal state, and then the search algorithm explode the state space. And how does it searches the state space it applies the move gen function which is neighborhood function which gives you the neighbors of each state, and then you inspect one of the neighbor and see if that is a goal state that you entrusted. If not then you generate more states and so on ((Refer Time: 02:04)).

So, this search state space search generates a search tree. The state space is at graph at least simplicity it exists even though has it with us. And the state space search generates a search tree, and we saw that this search tree is characterized by. So, we start with some search node, and then we go down searching for the tree we had seen this before so; obviously, every tree has two kinds of nodes; one kind of node is we will call as we have called as open. Open is the set of leaves actually the set of leaves we call as open, and the set of internal nodes we call as closed.

The open leaves are set of leaves is search frontier that is the set of candidate nodes that you have generated but you have not inspected. That we can think of the search frontier, and set of internal nodes is the memory of past nodes visited. So, I will just call it seen nodes. We started off by saying that open and close are sets, then because, you want to implement this using some algorithms we said let us call them list essentially. And as we will see today list notation is not necessarily I mean list structure not necessarily the base structure we will come to that little bit of a while.

But basically the search space is the search tree is characterized by these two let us call it as set or lists whatever, open and close essentially. And just recap the algorithm we extract. We had moved from storing only the states to a pair of states, and the pair consists of a given state, and the parent state. The parent state is the state from that given state was generated essentially.

So, we have this node space we said that initially open gets this pair of start comma nil. I will just write the outline today, and then, while open not empty. If it is empty then it will turn failure. When will it be empty it will be empty only for finite state spaces that you would have ended up by inspecting all the states. And none of them happens with the goal state in which case it will be empty, and then you can say that goal state cannot be found.

For example, in the eight puzzles I had pointed out that the state space is actually partition into two sets of states which are connected to each other. Once set is not reachable form the other set. If you gave the state say it in one set, and the goal state in the other state then it would not be reachable, and you would end up storing all inspecting all the nodes reachable from the start state and say that we know we cannot solve that problem.

If open is empty then you return failure that will happen at the end we will not try it here. Otherwise we get node pair as head of open I will not write the details we did it with a little bit in the last class, and otherwise you can look up the book. From this node pair we extract a node let us call it as n. which is the first element of the node pair, and we apply the goal test n we apply the goal test function. Remember the goal test function takes a node, and tells you whether it is a goal node or not. Somehow you implement the goal test. So, move gen n, and goal test n these are the two domain functions that we have the rest of the search algorithm that we are writing is independent of a domain. As long as summery provides with you move gen function, and the goal test function you can use this algorithm to solve problems in any domain essentially.

You do the goal test; if it is yes then you reconstruct the path. If it is no you apply move gen n. And you get some success you get this neighborhood this node n, and to this you do some filtering you remove the things that you have seen. So, I will just say remove seen there some function which will take the output of this, and filter out things we which are already existing in closed or in open essentially. We do not want to generate the same node again because if it is in closed we have already seen it, and if it inspected again we are likely go into loop. If it is opened we will see it sometime because, it is an open anywhere no point keeping two copies of it in the open. So, this move gen basically removes moves from open, and close the successor of n.

And then we have a function called make pairs. Whatever remains after this filtering we make pairs what is the pairs. This node we take this node n as the parent of each of these nodes inside this. This nodes will look like x coma n because, these all the children of n.

n should be the parent of these nodes, and then we add this things to open or to the tail of open. If you want to be precise because, if you remove the head we have not removed the head here. So, actually we should add ((Refer Time: 09:01)) but anyway that is a basic idea. And then we saw that there are two ways of doing this; one is as the stack, and the other as a queue. which means that you either add the new ends at the head of open then it behaves like a stack because, they will be first once to be inspected. Because we always extracting the node from the head of open. And when it is a stack we saw that this behaves like depth first, and when it is a queue we saw it behaves like breadth first.

What are the characteristics of depth first and breadth first? Depth first basically dives into the search tree. And breadth first is more cautious its set of plots through these things. These were the two characteristics of these two. You can say depth first just goes where it is nodes takes how to speak, and breadth first stays to close to start space essentially.

Then we had looked at the properties of these two algorithms. So, we want to compare properties on two features; one is time complexity. Now time is bad for both by bad we mean the worst case situation or the average case situation. In the best case they will find the goals for example, depth first search finding the goals state in this branch itself in which case it would find it in linear time a breadth first must finds the goals state somewhere here which is very close to start state. And it will find it very quickly essentially that is a best case.

On the average case, and the worst case this time complexity for both is of the order of b is to d, where b is the branching factor. And d is a depth at which the goal occurs essentially. which were there was a little bit of a difference the first breadth search had slightly moved time complexity then that of depth first search it was only slightly moves essentially.

This is something that we will start addressing today a bit later it is a time complexity we saw completeness. And by completeness you mean will it find the goal state or will it find a path to the goal state if one exists. And the answer in the case of breadth first was vocally yes in the case depth first it was it is guaranteed to find the goal state for finite state spaces not for infinite state spaces. Because in infinite state spaces they could go some infinite loop or some infinite branch. We will assume that we have working with finite state spaces and will answer yes to this.

But keep in mind that this is only for finite state spaces only. Then we saw two more properties one is space complexity, and we found that space was good for depth first because it keeps only it adds only consent number of nodes as it goes down because if the branching factor is b it goes down it will add b nodes inspect one of them then again add b nodes inspect one of them and so on. Adds b nodes which means this space required goes linearly.

Whereas breadth first search will first inspect the entire loop generate till then of all those nodes. So, it will become b into whatever the width of that was, and therefore, it multiplies by b as if goes down deeper and deeper and therefore, these goes exponentially.

So, this was the plus point for depth first search. But quality was the plus point for... I will just write plus here for breadth first search. Because, of the fact that breadth first search only floats slowly into the search space. At whichever the layer the goal space goal load occurs it will find that path till the goal load essentially which means it always find the shortest path to the goal. And I scope you have convinced yourself by constructing twice examples if we have not please go and do it essentially.

These are, this is what we did last time. There two things we want to do today one is try to see if we can find an algorithm which will combine these two plus points. And the other is to tried address this time complexity. Somehow, because you if you have an exponentially time algorithm nobody is going to buy it is essentially you can only solve very small problems with it not problems of significance size.

Let us first look at this is there an algorithm did anybody give a thought to this which will combine these two properties of depth first breadth first which means required linear space but, guaranty an optimal solution. I take it you are not been reading my book yet. Let us look at some variations of this all the algorithms that we are looked at today are blind search algorithms, which means that they have no sense of direction they always.

If given a state space if this is a start state and this is, and where ever the goal state may be in this state space the behavior of the algorithm would be the same. So, that first would go of in the direction back track try something else back track try something else and so on. Breadth first will go down and down and down till it expands the goal could be here the goal could be here or goal could be this side it does not matter from that essentially.

So, let me introduce one new algorithm or two new algorithms. One is he will call depth bounded. It says that variation on depth first search, and the variation is that we have put the depth bound do not go more than twenty steps go and go more than forty steps whatever some depth bound we have put and said. So, what have done we have cut of the search here some level and we had said that do depth first search on this truncated search space. What is the characteristic of this algorithm? Depth bounded depth first search it is linear in space why because, it is depth first search to start with its complete.

Who said yes? Why is it yes? What is meaning of complete? We said that if there is path see this depth bound is something that you have imposed it is does not come from the problem. The goal could have been some where here you know, what is the depth one? depth one says that it is like a ((Refer Time: 17:23)) do not go beyond this line. So, if you find the goal within that, yes you will get the solution. But if the goal happens to be outside that like here which could be somewhere here, and find the solution.

So; obviously, it is not complete. But, it is faced efficient because it is depth first search. Now, let us do a variation let us have an algorithm in which we say depth bound is equal to zero. So, this is new algorithm I am writing we initialized depth bound to, and then we say while goal not found do this algorithm let us call this d b d f s. And let us say this a depth bound d b. So, let me use d b here also. This is algorithm it takes an argument of course, it takes a start node and everything,,, but that we will assume is hidden or glover whatever.

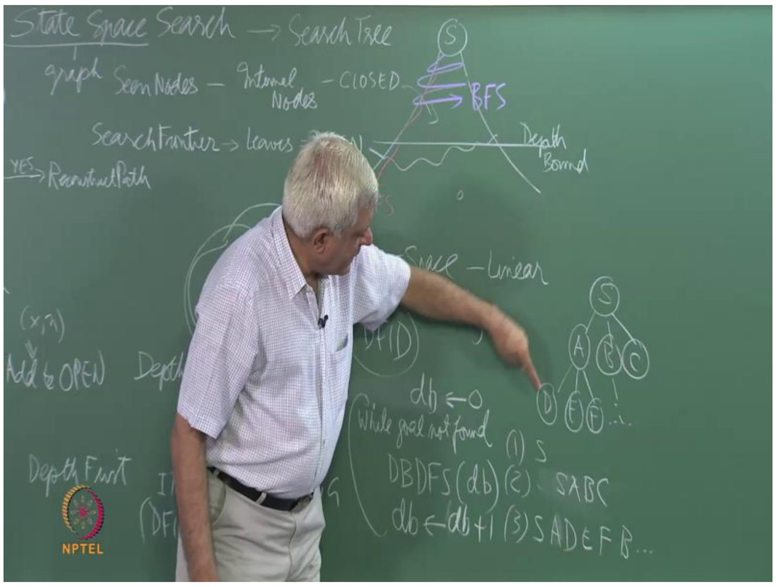
So, while goal not found do depth bound depth first search with the bound d b. So, zero means you just inspect the start node one means you go one level deeper two means go two steps deeper in that, and then you say. So, we have this put this in a loop. So, this a new algorithm what is this algorithm? Call it is very well know algorithm it is called. So, I will write it here in case in either space iterative as a algorithm suggests. So, this call this is call iterative deepening in every cycle you increase a depth bound by one, and then do a depth first search. So, iteratively you deepen this bound to which you will search. And because, we are doing depth first search this is called depth first iterative deepening which is popularly known as d f i d. This is algorithm d f i d that. So, first we should understand what the algorithm is we us doing a sequence of this depth bound d f f s starting with depth zero then going to depth one depth two depth three and so on.

What is the property of this algorithm? So, let us talk about will come to time complexity in a moment let us talk about space, and let us talk about quality these are the two quantities we are interested. In here because breath first gave us on optimum solution guaranty the optimum solution good on quality depth first was space sufficient requires only linear space. What about d f i d? Space complexity, when he says space complexity we mean the size of open that is the convention we have been following.

Same as d f s why for the simple reason that it is d f s of course, it is not one d f s it is many d f s 's every time you do a different d f s with a different depth bound,,, but you are doing d f s. So, space is linear. Is anyone having a doubt about this? You should clarify this at this moment itself it is just doing a series of depth first search inside every cycle in this loop it is doing one depth first search, but it is doing depth first search. So, it must be requiring space complexity of depth first search which is linear essentially.

What about quality? Not completeness, does it guarantee an optimal solution does it guarantee shortest path there is depending upon I have given the algorithm completely. I am asking the question that this algorithm does it guarantee you an optimal solution. What is a argument behind the this? Argument yes answer any one willing to ask talk about. Why does it? How does it guarantee? The answer is, what is the behavior of this algorithm? If you look at only the new no it is now what is this algorithm doing it is going to re inspect many nodes. So, let us say we have a search tree in which

(Refer Slide Time: 24:15)



We started s then in the first on we look at only s. Then in the second on we look at s a b c. Then in the third round we look at see this is d e f and so on. In this third round what is the order in which you. So, in the first round it should inspect only s in the first cycle in the second cycle, it would inspect in this order s then s a b c. In the third round it will inspect them in s then a then d then e then f then b and then whatever the child of b is actually.

So, in the first cycle it inspects only s in the second cycle does search only till this step s then a then b then c in third cycle it inspect this s I have not drawn the complete tree. But s a then d then e then f then the children of b then the children of c in that particular order depth first order. But now, if you want to mark the order in which it first time visits a node then, you can see that s is visited in the first cycle then, a b and c are visited in the second cycle then, d e and f are visited in the third cycle and so on.

It the order in which it is visiting you nodes, if you look at the order inside this red circle you can that s a b c d e f s a b c d e f this is the order of depth first search. And if it finds the goal it should have found a shortest path essentially goal. Because we know that depth first search in this order always level order as you all also call it always find the shortest path.

So, you has convince yourself I think that this algorithm behaves d f i d combines both the things that we desired which is that space should be linear which was depth first search and quality there you should guarantee the optimal solution which breadth first search. And this is actually giving it to us. You might say that this is actually sequence of depth first searches as a masquerading as a breadth first search.

Because, the behavior in terms of the path that it finds would be same as what depth first search have done. And because the very first level at which the goal appears this will terminate we know that it has found the shortest path. Any questions…

Student: ((Refer Time: 27:48))

What I written here is while goal not found. And So, I have lost over some detail if let us assume it is a finite graph it is a infinite graph it will keep searching. So, let us assume it is a finite graph then I will leave this as small excise for you to discover that at which point no new nodes has been added which means if you just inspect the next layer if there is no new node. Then you have inspected the complete graph, if you inspected the complete graph it should report failure. But till that point it should keep deepening and searching till finds a goals.

Student: as number of as a height of the tree till goals.

It is not a tree it is not a tree it is we do not know where the goal is that is a whole idea about the searching. We are in some space and we are exploring the space by using move gen function. And we were trying to find a path to the goal. So, first of all we do not even know whether a goal exists for example, in that eight puzzle I may give you the goal state as one which is not reachable. And secondly, we do not know where it exists at what level it exists. So, the whole idea is to search for the path.

You know this iterative deepening algorithm they were devised in a chess playing situation. And you know we saw when we looking at the history of e i that chess playing has long be fusion with e i people, and they want to make the programs play tournament under tournament conditions. And tournament conditions for those of you who play chess know that you have allotted certain amount of time for making a certain number of moves actually. So, the time available to the player is fixed essentially.

Now, we will see game playing programs later in these codes. But essentially they also explore tree of some kind they can also explode tree up to various levels of depth. The deeper the explore it so, exploring a tree basically means you make you explore combinations if I make this move then the opponent will make this move then I will make this move then the opponent will make this move and so on. So; obviously, this analysis you can do to any level till the end of the game in fact. But that is not really possible. You do this analysis and then try to judge which is good move to make.

Now, in chess playing programs if you are playing under tournament conditions you have to be a aware of how much time you have essentially. So, iterative deepening algorithms devised that situation that you learn the algorithm. And let it go deeper and deeper as long as time allows suddenly if the calling algorithm calling program knows that time is running out will say tell me the best move, and it will play the best move. So, we look at iterative deepening again all at least we will mention it again we look at game playing algorithms.

Now…

Student: ((Refer Time: 30:53))

So, you are saying why do not you do breadth first search. But the reason we are not doing breadth first search is because, it needs to store this entire. See, what is open list of breadth first search? The open list is something like this across this tree, and this is a shape. So, the search plenty of breadths first search. In fact, looks like this, and as he go deeper and deeper this is growing exponentially, we know that the number of nodes in the d essentially. So, we do not want to store breadth to d nodes that is a reason why are not doing breadth first search. So, we are doing depth first search. So…

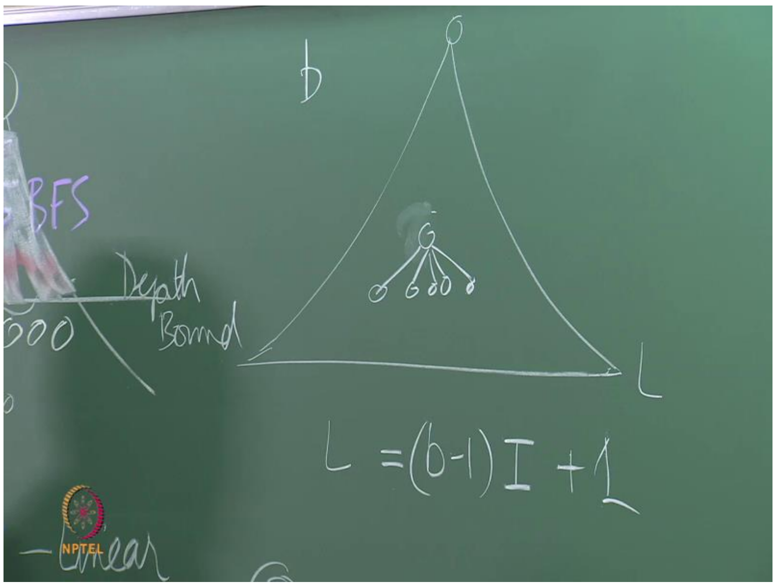
Student: ((Refer Time: 31:41))

Already visited but, what is the option availability you. So, if you think little bit about it you will see that see, I can see, what you are trying to say, you are trying say that, what is the point of starting with s all over again?. But, where do you start otherwise? In breadth first search you would have inspect you would inspected all these nodes, and then you would have inspected their children. They were the children they would be in open. But here we do not have them essentially anything. So, you need to give a little bit of thought to it yes that is a common doubt when we look at d f d f i d first essentially.

So, we are doing this extra work, what this extra work we are doing? We are doing a sequence of searches in which we are inspecting the complete tree at every level. First up to level s here then this whole tree then this whole sub tree then this whole tree and so, on. So obviously, we are paying an extra cost, is this cost worthwhile? What are the benefits we are getting. The benefits, we are getting is that we are getting linear space, and optimal guarantee solution guarantee of optimal solution the extra cost is going to be the price that you pay off inspecting all these nodes, which are not inside the this red circle again and again. So, s we are seen here again we are seen it here again we are seen it here.

We have seen here we are again seeing it here. B we are seen here this whole count is the count of measure of time complexity of d f i d because we are seeing all these inspecting all this nodes. So, how much is this extra cost, is it worthwhile? Is the question, what is your intuition? So, let me repeat in d f i d, we search up to some level let us say this level. We do d f d f i at some for some depth bound we come up to here and then to inspect these next level nodes, we search this whole tree again including this for depth. So, this shaded portion is the extra work we were doing for inspecting this new set of nodes how much is this extra work. Is it high or low? Let me just ask way simple question. So, we have go back to our study of trees and you might have done in data structures of some other course.

(Refer Slide Time: 34:15)



Let me take an arbitrary tree of branching factor b, and we will take a complete. So, for this argument sake we will assume that the tree is complete which means the every internal node has exactly be children for the sake of analysis which is not the case as we know. For example, in the eight puzzles corner when the blank in the corner there are only two moves that you can do? Whereas, when the blank is in the center you can do four moves so; obviously, the branching factor is not constant. But let us assume for the sake of analysis that branching factor is b and it is constant. So, that and this is the frontier that we are looking at.

So, which the set of as I said right the leaves of this tree are the is the search frontier and the internal nodes i of this tree are the nodes that we are visiting again for the sake of inspecting these l nodes. At any given stage for this is a depth first would have done depth first search just inspecting this l nodes, what is d f i d is doing? It is inspecting i plus l nodes that is a extra work it is doing the whole question is and depth is should give us insight into the nature of this monster that we ((Refer Time: 35:41)).

So, let me give you an nice argument of course, I am sure you have done this. In some course, what is the ratio of internal nodes to leaves in a complete tree? But, I remember in mathematics professor k Joshi from IIT Bombay had given a very nice argument. And you can visualize tournament which is going on. So, since branching factor is b we will assume that it is something like that say 100 meter sprint or something like that. And at any note there are b children. So, any search thing can be seen as one game or one race that you want to call, if it is binary I could have talked about tennis tournament but, it is not binary it is branching factor b. So, let us assume it is like a hundred meter sprint and b people compete in a heat. And only one selected form the top essentially. So, that is the nature of this competition.

So, there are totally l competitors and in every internal node i what happens every internal node is a heat. In every internal node one out of b goes head and the b minus one are eliminated is actually. And in the end of course, there is only one winner in the end, and all the rest are should I use the word loser or we should say also run may be I think that is a better word. So, if you give some thought to that you will see that the total number of parties' forms which is l which is a number of leaves i n a world tournament. So, these are also called winner trees you might have studied them somewhere, actually is equal to b minus one into i plus one. So, what is a argument for this? Of course, you can give a proof by indexation we are more mathematically inclined.

But this is this argument is just a tournament argument it says that in every internal node b minus 1 players are eliminated. So, if i is the number of internal nodes then b minus one into i is the total number of base which are eliminated, which is of course, l minus one and thus one winner who stands out. So, the total number of competitors is given by this and this gives us a relationship between i and l.

So, you can write i is equal to l minus 1 divided by b minus 1 and then you can compute l plus i divided by l which is the ratio that we are looking what which is the amount of extra work d f i d is doing as compare to breadth first search. Breadth first search would have inspected only l nodes only this boundary nodes d f id inspecting the entire tree it is l plus i node essentially. So, if you if you write this. Plug it and do a little bit of simplification you will see that this is for large l d over d minus one appropriately this the small factor somewhere with I will leave out.

So, what is what are we saying? We are saying that the amount of work which d f i d is doing as compared to breadth first search, which negligibly more essentially. Just d over d minus 1 times more essentially, and that is should not be surprising to you because, this is the nature of these breadths explosion. That we have b is to d nodes here in this layer, and all the internal nodes are b is to d minus 1 by b minus 1.

So, if you ignore that minus one next say for large or large branching factor you can ignore that you can that l is roughly b minus 1 times I, which is also what we have written here? So, the number of leaves as you go deeper down tree is b minus 1 times entire set of nodes that you seen before. Anything you did before feels in comparison to what you are doing at this level. So, what and that everything you did before is extra work which d f i d is doing, it just re seeing visiting those nodes again and again essentially. And if you go through this argument you will see that the time complexity of the d f i d is not significantly more than breath first search.

We had that seen that breadth first search was a little bit more than. So, b f s to d f s was b plus 1 over b or something like that i do not remember exactly, but I think b plus one over b. Breadth first search little bit doing little more over than depth first search, and d f i d is doing only little bit more work than d f i d. And what is that advantage we get? We allowed using linear space and we are guaranteed the solution. So, it is very nice algorithm think about this little bit i want to... Before we move on to this other question of, how can we get around time complexity? I want to address this question this thing that we did.

Remove seen, what is remove seen? Saying is that for every new child that you are generated or every new node that you are generating check whether it is already present in close. Let us assume that we have simple collection nodes and we do not have these node pairs and. So, on let us just ignore that for a moment. But we have collection of we want to check whether the given node exists in closed. What is the cost of this? actually or complexity are we paying a heavy cost for simple checking whether we have visited on node again. What would be an algorithm for doing this? So, what is the task? The task is given a new node n.

And given a list of nodes which you called closed which is the node that we had seen before, well I use the term list. But and it does not matter whether n exists in that set or list or not. What would be the algorithm for doing that?

Student: ((Refer Time: 43:22)) in which we can store a is not having a bit bit director in which we can store. So, it would be order one.

We will refine that in the moment. But if treated list how would i do it. I would have to sequentially search which would mean it would be linear in the size of closed and how was closed going? As depth close remember is a measure. So, that is why when we said that when we talk about time complexity we will appropriate with the size of closed. Which means the number of nodes that we are, what is closed? Closed is a node that we already seen.

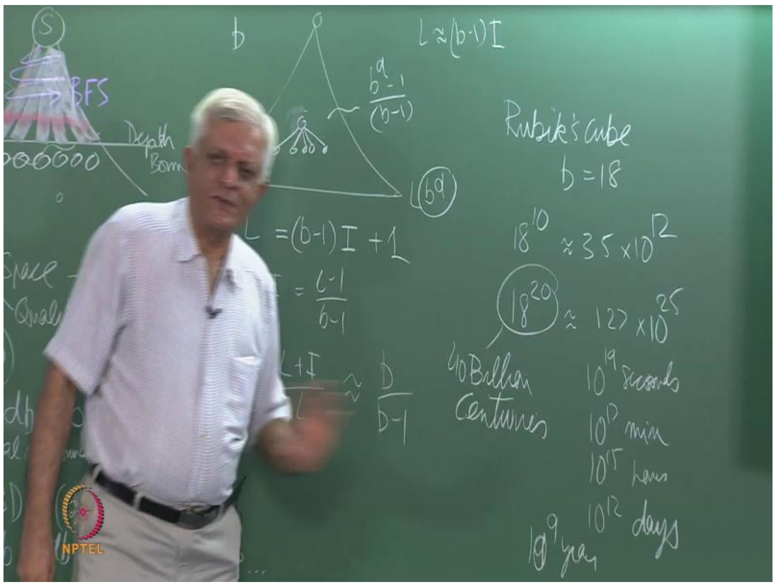
And we had said we appropriate it with the size of closed and there we had made an assumption that checking in closed is not expensive. When if closed is going to be expensive checking in close is going to be self if close growing expansible then checking in that will each of each time will take expansion amount of work then, it going to be a tuff thing correct. So, there is a i want to make a distinction between when conceptually we think of close as a list its fine as per as the problem solving algorithm that we are considering. But if you want to put on your software engineer or programmer had then, you have to be more concern about how to implement closed.

So; obviously, set is not good idea anyway sets you have to implement in some way or the other, list is good idea because, you will have to cancel the list. And we will take a queue from what she said can we do this faster. I am not saying that we should make in area of bit factors you should be bit louder, and bit more confident when you make that answers.

Student: ((Refer Time: 45:37))

While a binary search we would assume that the given set is a ordered set because you need to have a this thing. But a hash table is the perfect solution to this you must implement closed as the hash table. Even though we say it is a list and that can that is only for discussion purposes. But if you are going to implement the algorithm then close was be a hash table, and we know that on the average if you design your hash table well if your hash function in nicely chosen. Then it will give you average consent time there essentially. We come to open again in the little while. So, I keep talking about these as a monster and a beast. So, what is a size of this problem? So, let me give you some idea about this.

(Refer Slide Time: 46:38)



if you look at the Rubik's cube remember that b is equal to 18 and Rubik's cube is a nice example where b is constant at any given state you can make this 18 possible different moves. 3 for each face, and there are 6 faces if you had to search up to a depth of ten which means you want to explore the space up to depth ten. Then you would have 18 is to 10, and that turns out to be 3.5 into 10 is to 12. If you have searching for the Rubik's cube that what will happen if i make try all combinations of 10 moves that can do then you have to inspect what 10 is to 12 states essentially. And what is a typical length of a solution any idea of Rubik's cube problem is it less than 10 or more then 10. It is more than 10 most of the time if you have to search for depth 20 this an let me get I have the number somewhere is 1.27 in to 10 is to 25 how.

Long does it take it to inspect 10 is to 25 nodes you know we do not have a good idea of big numbers 10 is to 25, 10 is to 30 they sound same to us. So, let us say we assume, do some very rough calculations let us say we can inspect a million states in a second. So, we need 10 is to 19 seconds. Let us assume we do not want to divide by 6 multiply divide by 60 and all that let us assume that a 100 seconds in a minute. So, we will have 10 is to 17 minutes, and let us say there are 100 minutes in an hour then, we have 10 is to 15 hours. Let assume that there are 100 hours in a day.

Then we have 10 is to 12 days how many days is 10 is to 12 days let us assume that there are 1000 days in a year is 10 is to 9 years it is about billion years. If you had a machine which could inspect a million moves per second, and it had to explore up to depth of 20 according to this calculation it would take you about a billion years. But if you do the actual calculation, and I did it at home it actually takes 40 billion centuries to inspect 8 10 18 is to 20 nodes.

Is 10 is to inspect 10 is to 25 nodes it take about 40 billion. I will sure you are not willing to wait for so, long essentially. So, will try an address this how and I said people have more recently written programs which will find you the optimal solution in a Rubik’s cube essentially. How do they do it essentially? So, let us answering this question, how do you attack this exponential time complexity? Can you set of improve open that? Essentially. So, we will do that in the next class, and we will take a break now. And when we look at heuristic search essentially. So, we will take a break and come back in about five minutes.