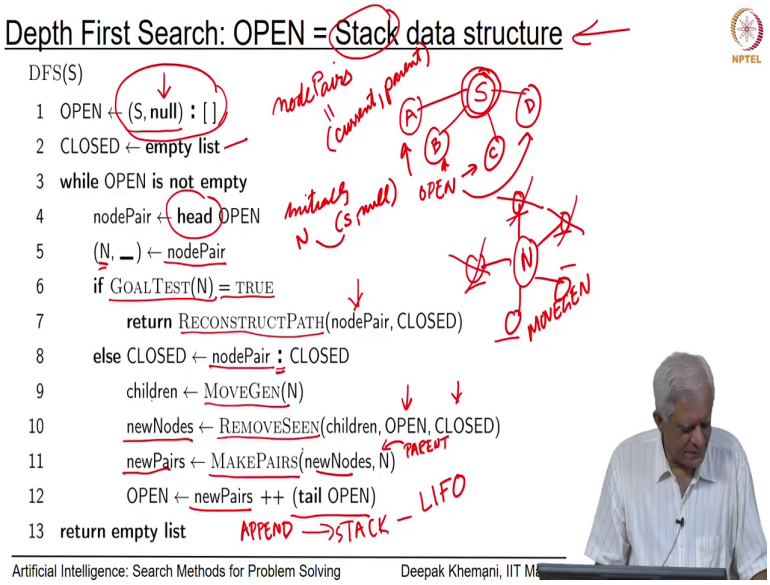
Artificial Intelligence: Search Methods for Problem Solving
Prof. Deepak Khemani
Department of Computer Science & Engineering
Indian Institute of Technology, Madras

Lecture – 20
DFS and BFS

(Refer Slide Time: 00:14)



So, in the last session we saw that we have devised a set of list manipulating operators, we have looked at set of list manipulating operators which give which are given to us by many programming languages and we can easily translate an algorithm written with those algorithms to a programming language.

So, let us look at my first deterministic algorithm, this algorithm is called depth first search and the algorithm is similar to what we wrote earlier except that everything is frozen, there are

no non deterministic choices here. As before the algorithm maintains two lists and it maintained two lists of pair.

So, both OPEN and CLOSE they will contain nodePairs. So, just to remind you nodePairs are pairs where you have current node and parent node and we start with creating the first node pair, the first nodePair that is available to us is the start node. So, if you look at the state space then this would be a start node and it would have some children and so on and so forth.

But since this is a start node we assume that it has no parent and in fact, the algorithm for reconstructing the path will terminate when we reach the start node. Because we will identify the start node by a node which does not have a parent. The name S is not meaningful as you know in programming languages you can give a name to any name to anything its a meaning that is important and in our case the start node will be identified by a node which has null as a parent node.

Now, remember that this is a list this thing is a list where the head is this nodePair which is S comma null and the tail is an empty list because we have not yet started exploring nothing. So, what would happen is that, once we explore the space then we would generate its children and then we would add these two OPEN.

So, this will go into open, this will go into open, this will go into open, this will go into OPEN and this will go into close, we would often the node CLOSE by double circles essentially, but initially as you can see CLOSE is an empty list because we have not seen any nodes here.

So, remember that the algorithm simply says generate and test pick a node test whether its a goal or not if its not the goal add it children to OPEN essentially. So, we will start of a initially in OPEN we have only S. So, when we say when we extract that node, we will get the node initially we will get this S comma null and this S is referred to by this variable N.

So, in this name N is the name of the variable that we are always picking for inspection in the node knows. In the first instance it would be S, in the second instance for example, if we pick

this node A it would be A, in this third instance it might be B, in the fourth instance it might be D C and D and so on.

So, whichever is the node we will refer to that is N and like we discussed in the last class, this is the first element of a pair and we are not really particularly bother about the parent at this stage, we would be interested in the parent only when we onto reconstruct the path to the this thing.

So, we are interested in this node N and we can access it as the first element of this pair which we extract from OPEN and we always as we said earlier, we always take the first node in OPEN and that is the head of the list and that is easiest to extract essentially. So, we take this node N and we apply the GOALTEST function as before we have done this in that non deterministic case, but let us also go over this algorithm again essentially.

So, this statement says that this is true GOALTEST N is TRUE. So, GOALTEST is a function we tells you whether you are at the goal node or not and it returns in terms of functional programming languages either TRUE or false and if it returns TRUE, then we are done and we will just call the function RECONSTRUCTPATH that we will see in the following slides.

And to call this function RECONSTRUCTPATH we give as input the nodePair that we have just extracted of which the first element was the goal and whatever is enclosed which contains records of everybody's parents essentially right every node that we have seen and its parents.

So, if its not the goal node. So, if GOALTEST is true then we construct the path, if its not the goal, then the first thing we do is we add this nodePair to CLOSE because you know it may be leading to the eventual goal and we want be able to trace the path right and we can add it to CLOSE node simply as at the head using the same dot operator, then we call the MOVEGEN function which the user has given us which gives us the neighbors.

So, this is N and you get the neighbors of this thing and this is what MOVEGEN returns to us and we call this set of nodes that this returns as S children. From this set of children to these nodes we will remove any node which is already on OPEN and already on closed.

So, we have seen earlier the impact of removing these nodes is that you do not want to keep visiting the same node again and again, it only increases the search effort and you can often get away by simply not looking at those nodes again. So, there is a function that we will write and we will do it in the next few slides which is called the REMOVESEEN function and what this function does is that it takes its children which are generated by MOVEGEN function and removes any children which either already on OPEN or already on CLOSED essentially.

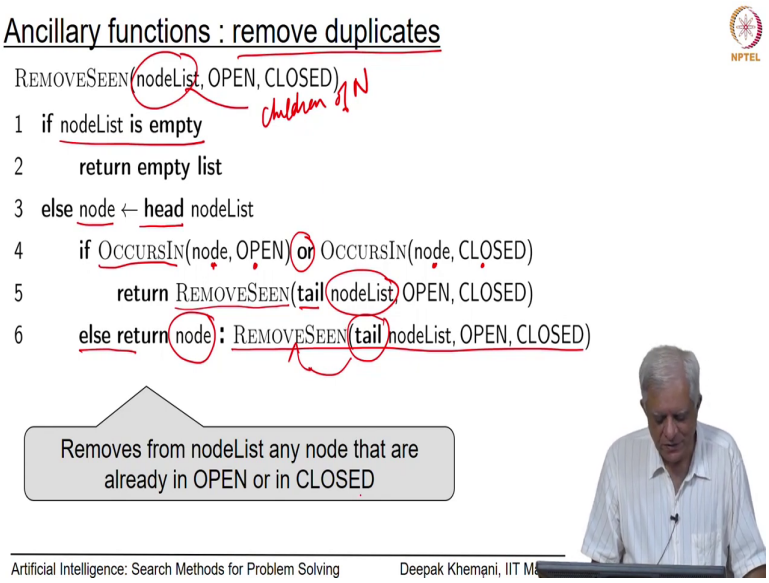
And so, now, we may have some number of neighbors and, but these neighbors are just the nodes that other neighbors have a search algorithm once nodePairs and we have this last small function that we will write, which will actually construct the newNode pairs. So, it will take this newNodes which we got after removing all the duplicates and to this newNodes for every node in the newNode we will keep N as the parent because all these were children of N essentially.

And having constructed this newPairs we will simply add newPairs to the head of this thing. So, we will append these newPairs at the beginning of the new OPEN why do we do use tail OPEN because we have not explicitly removed that nodePair from OPEN. So, we just implicitly say we will work with the tail and then these new things will be added at the head.

So, when we add this at the head of OPEN then we are treating OPEN as a STACK essentially because this is like a STACK and the properties of the STACK if you remember our last in first out and the latest nodes that are generated they would be inspected first and we will look at this how this behaves in a little bit later.

So, let us now fill in the functions that we talked about the RECONSTRUCTPATH function the REMOVESEEN function the MAKEPAIRS function and so, on and if you do that then the algorithm would be complete.

(Refer Slide Time: 08:51)



So, first let us focus on the remove duplicates function. So, what is the task that we are given to us a nodeList? What is this nodeList? These are children of or neighbors of N and from this we want to remove anything which already OCCURS in OPEN or CLOSED.

Now, remember that OPEN and CLOSED are both made up of nodePairs. So, every node in OPEN and every node in CLOSE is the pair and we will need to look inside that pair to check whether any of these children are present in that or not essentially.

So, here is a simple algorithm which does this, its a recursive algorithm and like any recursive algorithm we first look at a base clause and the base clause is that the node list is empty if it is empty we are nothing to do we just return the list essentially.

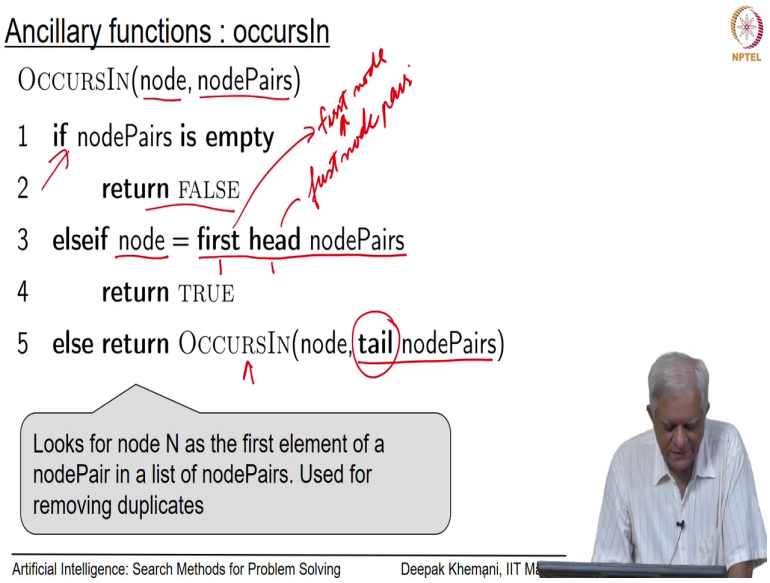
Otherwise we take the head of the nodeList and we call it node we assume that we will write a function call occurs in which we will look at whether this node which is the head of OPEN that we just extract it occurs in OPEN or not and likewise we will use the same function occurs into see whether this node occurs in CLOSED or not essentially.

So, if either of same is true if the node occurs in OPEN or the node occurs in CLOSED then we do not want to add it. So, what we return is simply REMOVESEEN apply to tail of the nodeList essentially.

So, that means, we have taken look taken out the first element inspected check whether its then OPEN or CLOSE or not and if its not or it or if it is either in OPEN or in CLOSE then we just discarded and work with the rest of the list. If it does not occur which is when we come here to the else clause then we add this node to whatever we get by processing the tail of the nodeList and applying the REMOVESEEN function to the tail of the node

So, in that manner in recursive manner we go over this nodeList one by one keep checking whether its an OPEN or CLOSE or not the checking is done by this function called OCCURSIN which we will see and eventually remove anything which OCCURSIN either OPEN or in CLOSED.

(Refer Slide Time: 11:18)



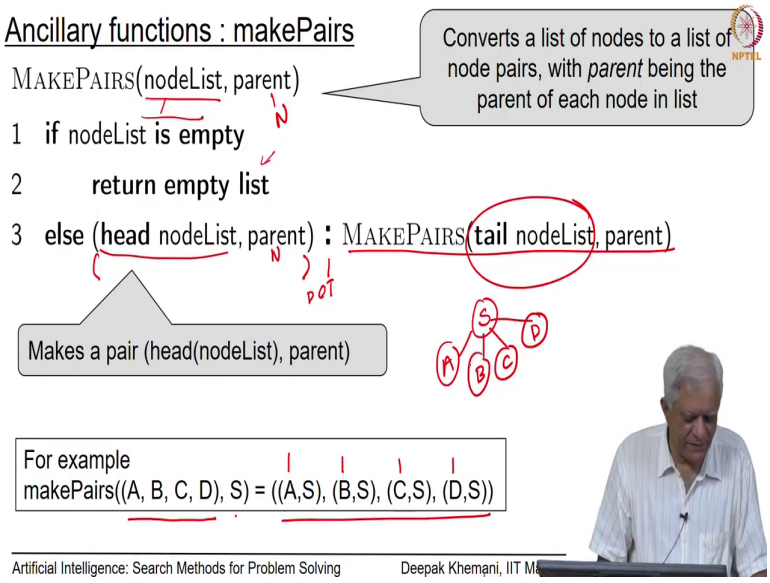
So, what is this function OCCURSIN that we talked about? What this OCCURSIN needs to do is to take a node as an input and a list of nodePairs remember both OPEN and CLOSE are list of nodePair and tell you whether this node occurs inside pair or not.

So, if the nodePairs is empty; that means, we have finish inspecting all the things, we just simply return FALSE else we look at the first node and if this node that we are interested in is equal to the first node of the head of the nodePairs.

So, this head will give you the first nodePair and this first will give you the first of this ok. So, the head takes out the first nodePair and this first takes looks at the first element of that and if that element is equal to node then we can return true that yes it occurs in that nodeList that we are looking at.

If it does not then we look into the rest of the nodePair which is given by the tail of the nodePairs and we use the same occurs in function to check for that. Eventually it will finish looking at all the nodePairs and it will come back to the base clause which is say that if they are no more nodePairs then the node does not occur inside that.

(Refer Slide Time: 12:49)



So, remember that once we have remove the duplicates, we want to MAKEPAIRS why do we want to MAKEPAIRS? Because our search algorithm denotes everything keeps everything as a pair essentially. So, every pair is a current node and its parent node and we simply want this function to do that essentially.

So, what is nodeList node list is the set of children of node N, the parent of all these nodes is N and for every and from this children we have already remove the duplicates which will. So, if you go back to the algorithm in c first we extracted children here in line number 9 then in

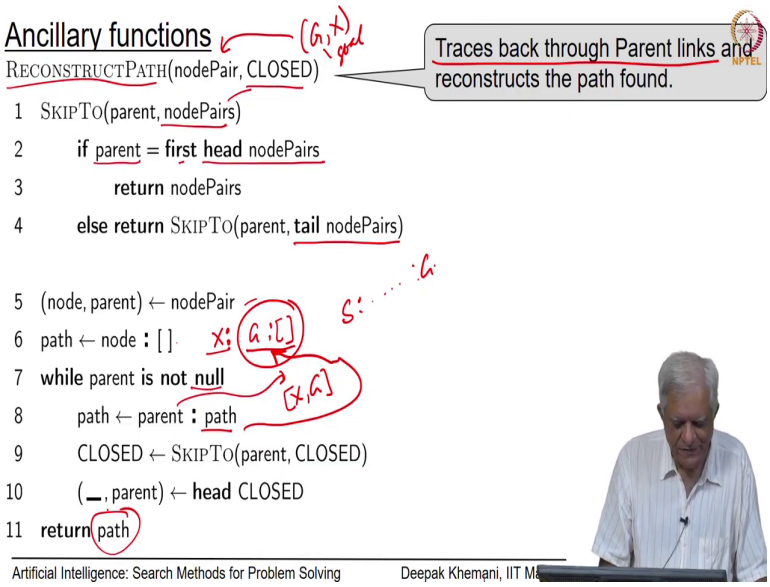
line number 10 we removed anything that is there and now we are focusing on line number 11 which says that you need to make these newPairs essentially.

And that is a function we are looking at and that function we will call as MAKEPAIRS and the make pair is follows that if this nodeList is empty, then we are done essentially we do not need to make any more pairs and the we can return the empty list.

Else what we do is we take the head of the nodeList which will be an element which will be a node, which will be a state in the state space and the parent of course, we have N and we make a pair out of this pair is made by using this brackets here and we add that to the head. Remember this is the dot operator which takes which makes a list which distinguishes the head and the tail of the list and we do the same thing for the tail of the nodeList ok. So, recursively we traversed on the nodeList constructing the nodePair one by one essentially.

So, for example, if the call to MAKEPAIRS is with a list containing A, B, C, D and the other list is S. So, remember that this is what we had said that if you are doing with the start node, then you have a as A child, B as a child, C as a child, D as a child. So, we have just four children which are A, B, C, D with and then we have the parent S what we want to do is to construct a list of pairs here where every element A, B, C, D is there along with S which is the parent of all these nodes as you can see from this diagram.

(Refer Slide Time: 15:27)



So, the function MAKEPAIRS does that for us. When we reach the goal node we want to reconstruct the path. So, we have seen earlier that the path can be reconstructed by because when we reach the goal node we will call it at the point when we are looking at a pair which is called G comma X or something where G is the goal. So, once we find the goal node we will say let us trace pack the path we know that the parent of G is X, but what is the parent of X that we have to go into the CLOSE list to see.

And therefore, the reconstruct path. So, this is the nodePair that will be input to this RECONSTRUCTPATH function and the other thing would be input is the CLOSED list and the CLOSED list will have somewhere in it a pair which says X in the parent of X and let us say the parent of x is y and then somewhere it will have a list pair containing y and the parent of y and so on and so forth till we reach the start node whose parent is null.

So, let us see how we can do that. So, essentially we are tracing back the path through parent links and parent links are kind of implicitly stored in the CLOSED list. So, let us look at this function, we want to go to that node in this list of nodePairs which is in CLOSED, it says that the parent that we are interested in is the first of the head of the pairs.

So, we look at the head of the nodePair and the first element is equal to the parent then we say yes this is a nodePairs that we are returning it, else we move to the tail of the nodePairs essentially. So, essentially we are scanning through the CLOSED list looking at one elementary time and seeing whether the parent that we are interested in which is this node x here is present there or not.

So, once we find that x then we will look for the parent of x and so on. So, what do we have we have node parent as the nodePair as input and the paths that we start with is the node followed by tail essentially.

What does this translate to? If you look at the example here when we are terminated with G comma X, then the path will look like a list which contains only G which is equivalent to saying G comma the empty list essentially and then when we find the parent of G we will put it here and then so on and so forth essentially of course, this we do not need the.

So, first we. So, first. So, essentially we are saying that we start of initializing the path to node followed by this thing which is this empty list and then when you find the parent of G we will add it and then the parent of X we will add before that and so on. So, we will essentially start eventually have a path which goes from S to G and so on essentially.

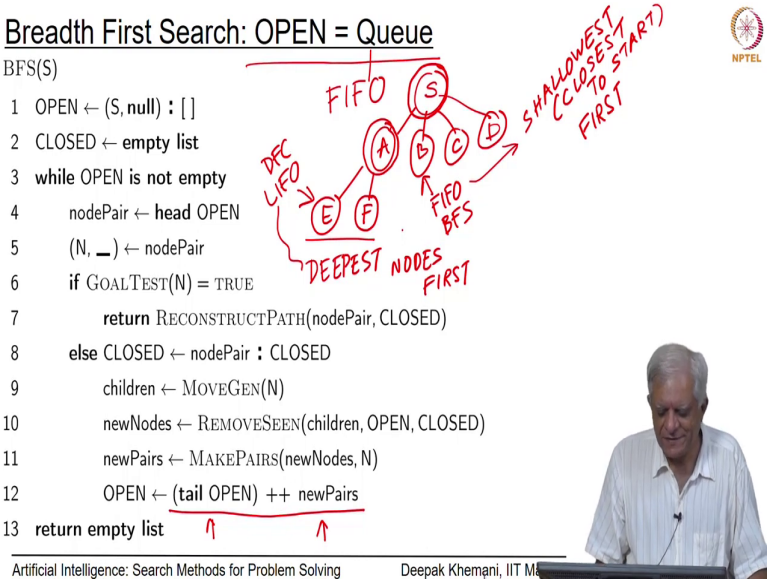
So, while the parent is non null remember that this will be null only when we have reach the start node, we add the parent to the path. So, what does this mean here? So, now, we our list will look like X comma G, where X is the parent and the path was originally just G.

So, we add that to the head of this thing and then look for where X occurs using this SKIPTO function that we just wrote and then keep repeating this process till we reach the null node

when it is when it reaches the when the parent becomes null, we know that we have found the path and then we can return the path essentially.

So, that is a function set of function that we have used in depth first search. So, now, we have a complete and detailed description of the algorithm depth first search.

(Refer Slide Time: 19:39)



Let us look at a variation of that first search which is where we treat OPEN as a Queue as opposed to a stack when we talked about depth first search we said OPEN is like a stack and the stack has a property that the last nodes to go in or the first ones to come out of open. So, LIFO property and in some sense you look at the most recent nodes first essentially. When we treat OPEN as the Queue has the property of first in first out which means that the first nodes to go in or the first ones to come out.

So, if you just look at the example that we have been talking about if you have S and A and B and C and D. So, when we have put S into CLOSED and A B C D are in OPEN the head of OPEN is always A. So, we will always put A into CLOSE and we will generate the children of A let them be E and F let us say. Now the question is after we are inspected a which is the next node that we should inspected.

Now, if you remember how depth first search works, it says the newest nodes first or the last nodes which we are inserted into OPEN first. So, that was last in first out. So, the last in first out would be picking the node E, but first in first out would pick the node B because this nodes E and F would have gone to the end of the queue and this is doing this treating OPEN as a queue essentially.

So, depth first search would pick E and breadth first search would pick A, we will look at the behavior of this algorithm in a little bit more detail the do we will do a comparison of which one seems to be better, but the fundamental difference between the two algorithms depth first search and breadth first search is only this. So, all this code that you see here is identical to what it was in depth first search, the only difference is in line number 12 and in line number 12 we have appended tail of OPEN to the newPairs that we found.

In depth first search newPairs came before and the tail of OPEN came later, in breadth first search tail of OPEN comes first and the newPairs go later and that is why it behaves like a queue essentially.

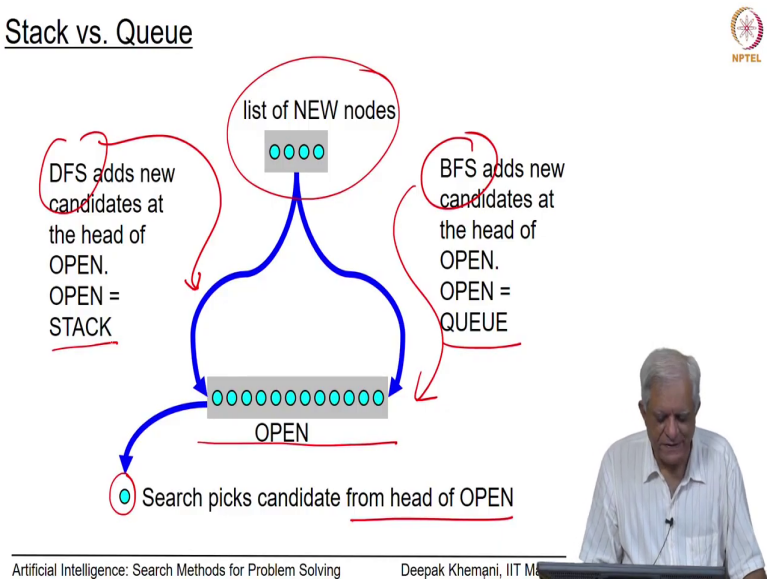
At this point we can also observed that because depth first search which was E and F what this does is deepest nodes first and that is why as you can guess it is called depth first search whereas, what FIFO does is shallowest nodes first or you might say closest to start. We will see in the next few slides that the consequence of following the shallowest nodes first will be that breadth first search will always give us the shortest path.

So, as you can see from this very quick comparison of the two algorithms, depth first search is like an impetuous search algorithm it just runs of an some direction and will backtrack only if

it reaches the dead end in that direction. Remember that if for example, E and F for to be dead ends then it would remove them from OPEN and add nothing to OPEN. So, eventually it will go to B and so on.

So, that is equivalent to saying that it has backtracking and so on. Breadth first search on the other hand is very conservative in the sense that it never trace far away from the start source node or the start node and stays that S CLOSE to the source as possible essentially.

(Refer Slide Time: 24:20)

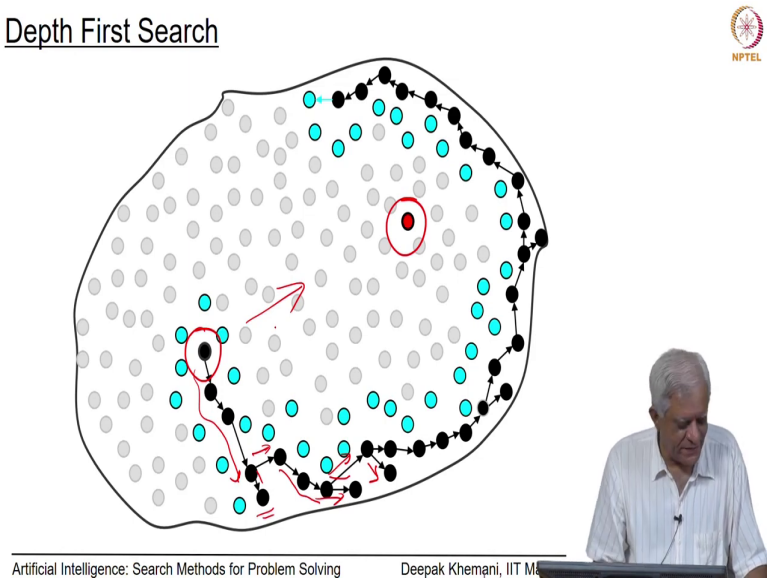


So, let us look at some of these properties in a little bit more detail. So, what we just discussed here is that this is the OPEN list that we have its ordered from left to right, we extract the first element from OPEN from the head of the list from head of the OPEN and then we call the MOVEGEN function which gives us as less list of NEW nodes and what DFS does

is that it adds them to the head of OPEN and what BFS does that it adds them to the tail of OPEN.

So, OPEN is like a queue for breadth first search and OPEN is like a stack for breadth first search and consequently the behavior and by behavior we mean in what order does the search algorithm explore the space is radically different essentially.

(Refer Slide Time: 25:12)



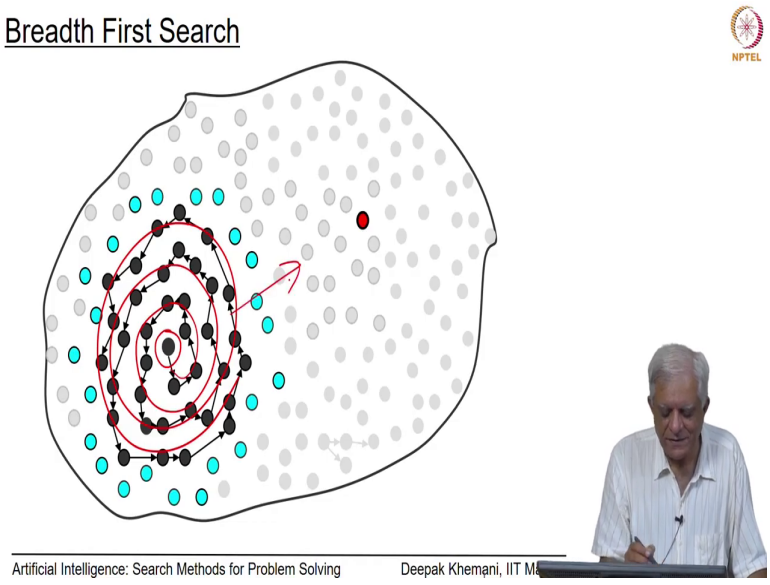
If you look at depth first search then you can see from this diagram that its gone off in some direction. So, it came down this path and then when it reach the dead end its backtrack one level, tried another path then again dived into some path then again reach the dead end. So, again it backtracked and again it tried different path.

So, its keep doing that. It keeps going away from the source node as far as possible and only when it hits the boundary it backtracks and try some other path and in this particular thing presumably the search algorithm would be like this.

Now, notice that this was the start node and this is the goal node and this algorithm has no clue as to where the goal node. Eventually of course, it will hit the goal node because it the state space is finite and it will inspect every node one by one because in every cycle of the algorithm it takes one node from OPEN and adds it to CLOSED.

So, if there are a finite number of nodes, it will eventually reach the goal node, but the path that it finds may not be the optimal one and that we will see.

(Refer Slide Time: 26:27)



Breadth first search on the other hand as I said tends to say CLOSE to the goal. So, the start node and then it just goes trying to stay as CLOSE to start as possible and then of course, eventually it will reach the these thing.

See as human beings if you were to think of if you were to think of this as a city map of course, we have not drawn the road, but you can imagine the roads here and every node is the junction of roads.

As human beings we could say we have a global picture and it would be nice if the search were to head in this direction somehow and not you know go all over the place or in this case also not stick CLOSE to home, but it would be nice if you were to go in that direction. We will make this attempt later after we are done with these basic search algorithms and those algorithms we will call as heuristic search algorithms.