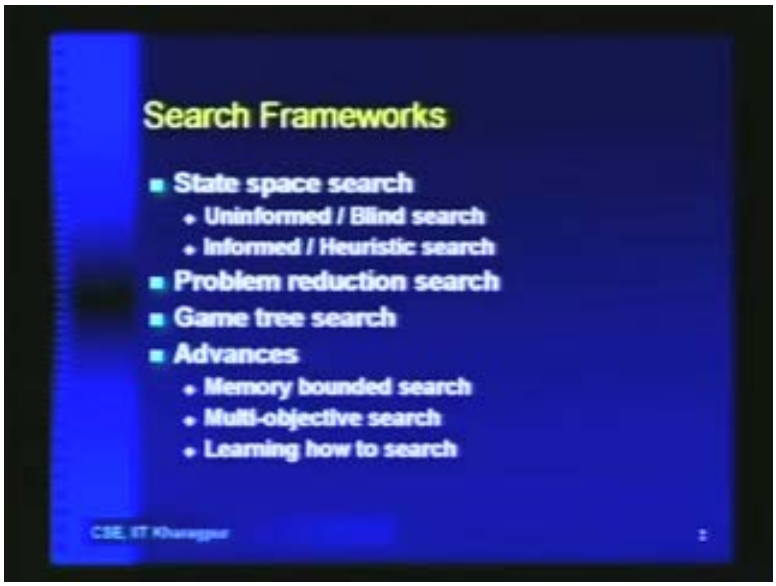
Artificial Intelligence
Prof. P. Dasgupta
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Problem Solving by Search
Lecture – 2

Today's class is the first chapter that we are going to take up in this course. The topic is problem solving by search. Slides, please. Under problem solving by search, we will look at several different search frameworks. The first is what we will call state space search. Here we will see how a real world problem can be mapped into a state space, and then we can have search algorithms for searching out the goals in the state space. Under these, we are going to look at 2 different types of search, namely uninformed search or blind search: these are search problems where we do not know any domain specific information, so we have to search without any additional knowledge; and the second thing will be informed or heuristic search, where we will have functions to guide the search procedure. And we will also study some algorithms for informed heuristic search. The next topic that we will study is problem deduction search. This will be done in the next lecture, and this will talk about problems which can be decomposed into sub-problems, in a flavor similar to dynamic programming, except that we will try to understand through search, how best to solve the problem.

Take for example integration by parts. You remember integration by parts? We can solve the same problem in different ways. So, we will associate a cost function with every kind of solution, and then try to see what is the best way to decompose the problem into parts and solve it the best way, okay? That is what we will learn in problem reduction search. And finally, we will have a look at game tree search, to give you an idea how about how chess playing programs work. We will have a few lectures on search in game trees and see how you can do optimization in the presence of an adversary. And I will briefly touch upon some advances in the search area, which is mostly research d1 in our department by our AI group. This is memory bounded search, multi-objective search and learning how to search. So these will be I will just briefly touch upon what these topics are.

(Refer Slide Time 03:17)



Right. So, coming to state space search, the basic search problem is as follows: we are given 4 tuples. The 4 tuples In the 4 tuples, the first is S, which is the set of states which is implicitly specified. Now this is very important. If you have a graph and you are asked to find out the shortest path from 1 node to another node in a graph, well, we know that we can do it by Dijkstra's or any other shortest path algorithm. And the problem is not hard. We have polynomial time algorithms for doing that. But in AI, we are going to look at problems where the graph is not explicit. And I will shortly come to some examples over that, where the graph is specified in terms of a start state s which we have here- the start state s and a set of state transition operators called O. So, if you apply the state transition operators on the states that will give you the next states of the current state.

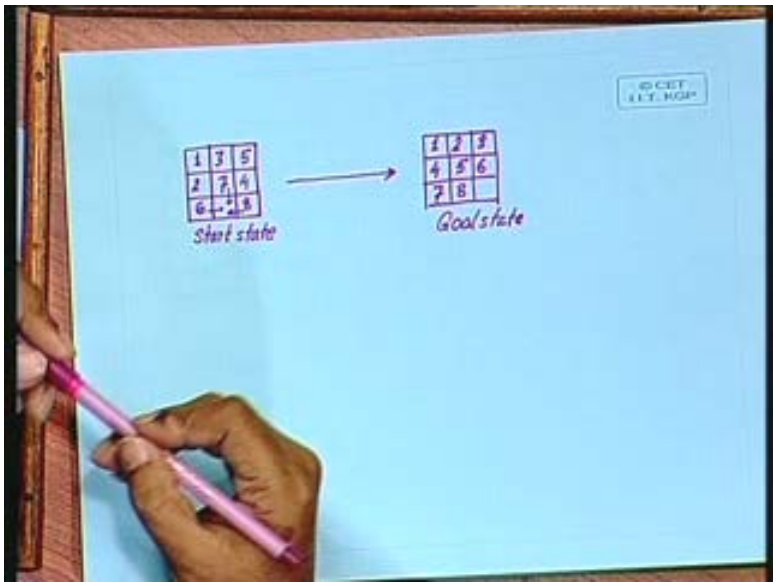
It is like you studied in switching. When you specify your finite state machine, you can either specify the whole finite state machine graph or you can actually just specify the state transition function or the next state function. When you specify just the next state function, then the actual state space is exponential in size of the next state function, right? Suppose I have a next state function for a 4-bit state machine. Then, you can potentially have 2 to the power of 4 states, but that does not mean that the next state function is of size 2 to the power of 4. We do not actually present the truth table of the next state function. Instead, what we have is a function which specifies the next step. Similarly, in AI also, we will have lots of problems where the search space is implicitly specified and the state transition operators will enable us to unfold the state space. We are also given the set of goals which will again be specified in terms of some formula and our objective is to find a sequence of state transitions, from the start state s to a goal state.

(Refer Slide Time 05:47)

So, in the basic problem we are satisfied by any goal state. Later on, when we go for optimization problems, we will look for the best goal state, right? Okay. The copies of these transparencies are in my website, so whatever you want to write down, you can always download from there and write it down. Let me take up a few examples. The first example that we will take up is the 8-puzzle problem. How many of you have seen the 15-puzzle problem? Okay, let me just briefly explain. I think once I tell you what the 15-puzzle is, you will remember. The 15 or the less say that the 8-puzzle, is a puzzle where you have a square grid having nine spaces, and there are tiles in each of these spaces. Say I have and these tiles are numbered from 1 to eight. Suppose I have 1 here, 3 here, 2… right? And 1 of the places is a blank, right? And this is a sample start state. Suppose we want to reach some goal state and let us say the goal state is the 1 where all these numbers are properly arranged, all these tiles are properly arranged.

Suppose this is the goal state. And now we want to go from here to here, and the way to do that is slide the tiles. So, you can slide this tile here, or you can slide this tile here, or you can slide this tile here, right? And keep on this doing this procedure, until you have been able to arrange all these tiles in the final form, right? You are not allowed to pick up a tile and move it into another place; you just can move them by sliding the tiles. So, the version of this problem with 15 tiles, which is a 4 by 4 square, is the 1 which is most popular. But because that is much larger problem, we are just talking about the 8-puzzle, where you have 8 tiles and 1 blank. Let us see- how do we formulate the state space of this?

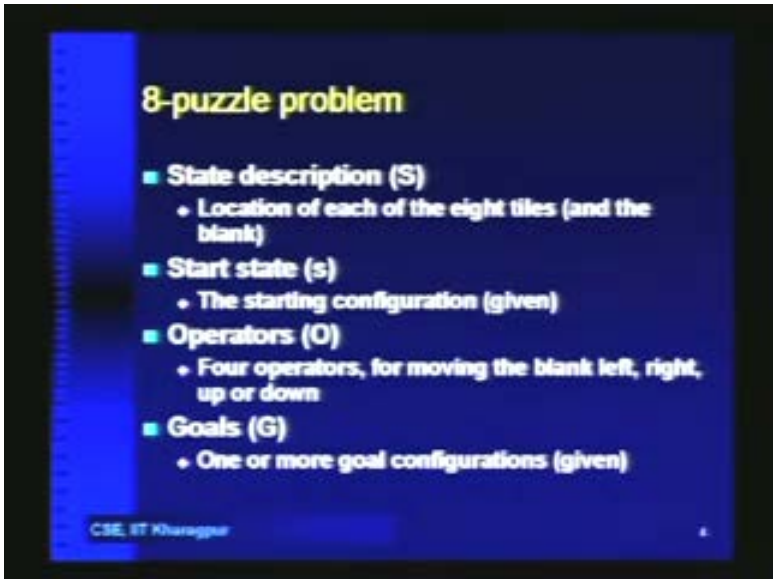
(Refer Slide Time 08:41)



We can say that the state description is the location of each of the 8 tiles and the blank, okay? Now, if you note here, that if we just indicate the position of the blank and the tiles, then what are the operators that can operate? Shift right, shift left, of? Yes, the only operators that we need to consider are the 4 operators for moving the blank left, right, up or down. So, if you just indicate the way in which the blank moves, that is sufficient. For example, from here, the blank can move up, it can move left, it can move right. It cannot move towards the bottom, because we will fall off the end of the puzzle. Right. So, if we just indicate the motion of the blank, that is sufficient to indicate the possible state transition operators, right? Here we have out of 4 operators; not all of them are applicable on each state. In a given state, it could be that 2 operators are applicable, 3 operators are applicable or all 4 operators are applicable. The search paradigm will start from the start state, which is the starting configuration, and try to use these operators systematically, so that we reach 1 of the goal configurations.

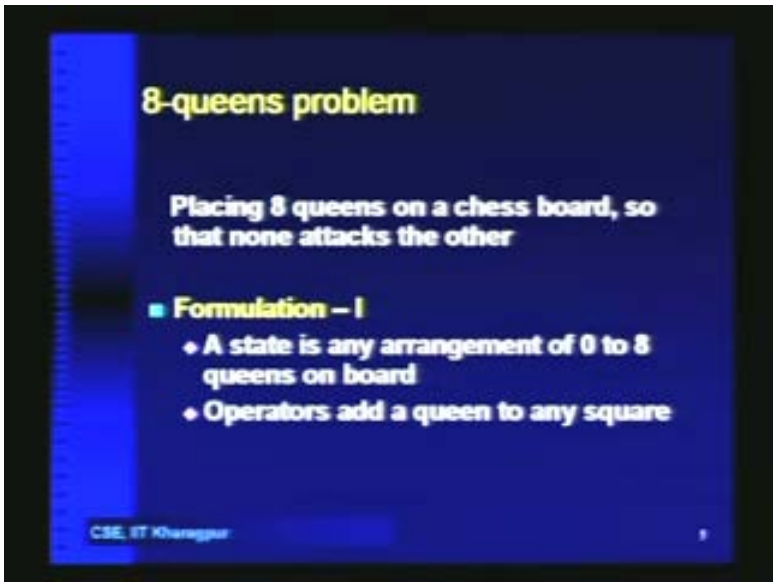
In this case, as we said, that the goal configuration could be the set of tiles arranged in the proper sequence. It could be any other goal configuration. Interestingly, not all goal configurations are reachable. For example, in 15-puzzle problem, if you physically pick up 2 tiles, and exchange them, then the configuration that you reach cannot be reached from the original configuration. Suppose I start from configuration x and I just swap the positions of 2 tiles- any 2 tiles- and try to reach this configuration, instead of swapping by sliding tiles. Well, you cannot do it. Actually, the state space is partitioned into 2 parts. You can reach half of the states by sliding this from this configuration and the remaining half by sliding from the other configuration, right?

(Refer Slide Time 11:29)



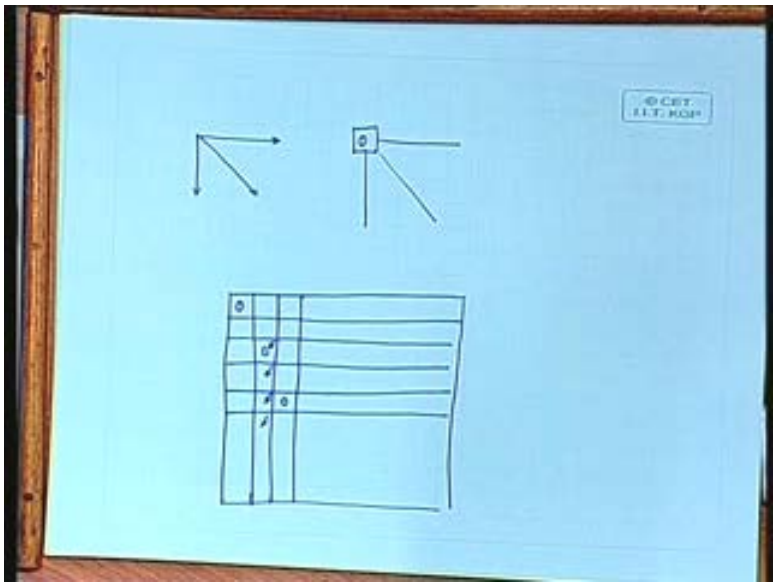
Let us look at another example. This is the 8 queens' problem, which I am sure most of you are familiar with. 8 queens' problem is placing 8 queens on a chess board so that none of the queens can attack the other. Everyone is familiar with chess, right? So, a queen can move horizontally, vertically and diagonally, so which means that if I put a queen on 1 of the grid points, then I cannot put anything in this same any other queen in the same diagonal, in the same row and in the same column, right? So, I have to put the 8 queens on the chess board, so that none of them attack each other. Now, if we look at the state space formulation, there are many ways in which we can do this. I am going to give you 3 formulations. The first formulation is that the state is any arrangement of the 0 to 8 queens on the board, right? So, there is no systematic way of putting the queens. I will just put 0 to 8 queens on the board. Now, what I mean by 0 to 8 is that it could be any number. So, I have placed 4 queens, let us say, and the next step is to add a queen to any square, right? This is an unsystematic way of doing it. I take the first queen, place it anywhere; take the second queen and just place it to any other square, where it does not attack. Now, I am not going row-wise or column-wise or anything like that. This is formulation number 1. Clear?

(Refer Slide Time 13:27)

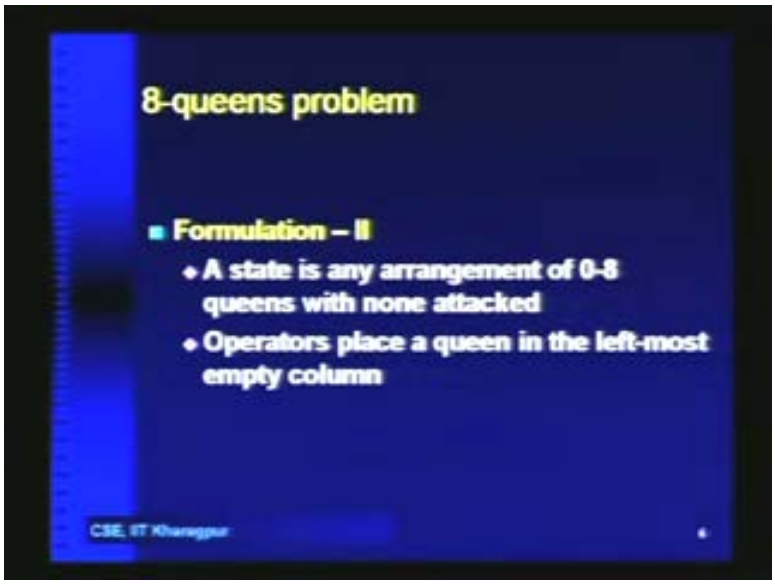


Formulation number 2 is a state is any arrangement of 0 to 8 queens with none attacked, but the operators will always place a queen in the left-most empty column. What does that mean? It means- suppose I start with the chess board. So, I will place the first queen in the first column. Suppose I place it here. Then the next queen what the operators will try to do is- the next queen will be put in the second column, in a way that it does not attack the first queen. So, we can put it, say here, for example, right? Then, if we again apply the operator, the third queen will be placed in, again, some square which is not attacked by the first 2. In this way we continue, and this is the state space definition. But you can also- instead of placing the second queen here, you could have placed it here also, or here or here. These are all the different operators to place the queens, right. But after placing the first queen, when I am applying the operators, I will never attempt to put a queen in the third, 4th or fifth column until I have put a queen in the second column. We are going left columns left to right, okay? This is formulation 2. It is slightly more systematic.

(Refer Slide Time 15:10)



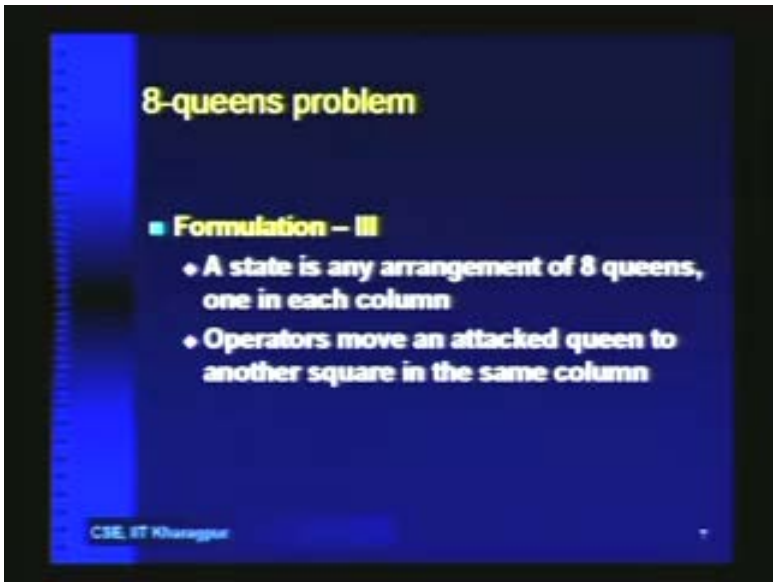
(Refer Slide Time 15:18)



Right. Formulation 3 is where a state is any arrangement of 8 queens- 1 in each column. So, I just without looking at any of the constants, I place 8 queens on the board, 1 in each column. They will attack each other, and then the operators move an attacked queen to another square in the same column, right? Here, the formulation is slightly different. In the previous 2 formulations, we were maintaining a valid partial solution and trying to grow it into a total solution, right, and this formulation is what we call an iterative refinement or iterative corrective procedure, where we just start with any formulation, any arrangement of the queens, and try to correct them based on the constraints. So, we will study some kinds of search algorithms which work in this way.

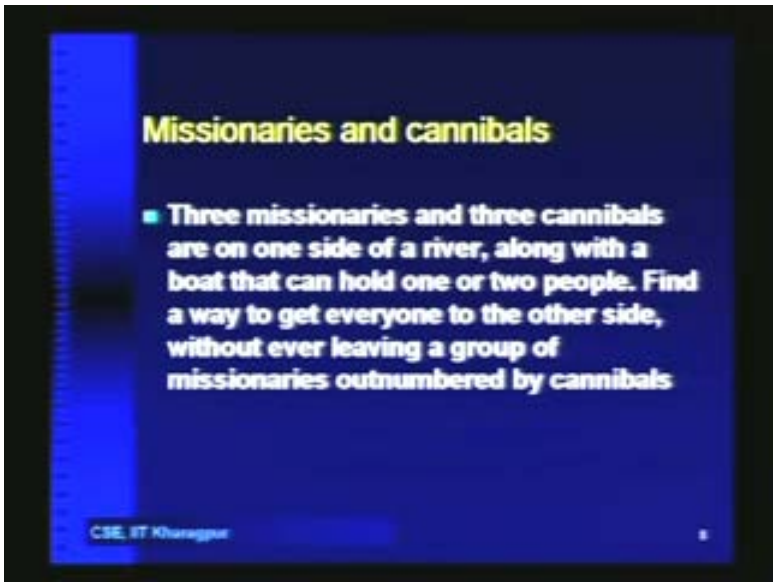
(Student asks question.) In formulation 3 also, you will definitely get a solution right, but it may so happen, that when you move an attacked queen to another square in the same column, you will move into another place which is also attacked, and then keep on doing this shuffling, until you reach a configuration where none is attacked. If you just think over it you will see that even this formulation is complete. That is, it allows you to visit all the valid goal configurations, but again, I have just shown you the formulations here, I have not told you how to solve them. Next thing that we will look at is- how do we actually go about, to systematically solve such problems? But the kind of formulation that you do is very important. If you can do a good formulation, then your search effort will be less. If you do a bad formulation, your search effort will be more. Right?

(Refer Slide Time 17:42)



The final example is the famous Missionaries and Cannibals problem. So, the story is like this- that you have 3 missionaries and 3 cannibals on 1 side of the river, along with a boat that can hold 1 or 2 people. Now, you have to transport all the missionaries and all the cannibals to the other side, but if, at any point of time, the number of cannibals are more than the number of missionaries, then, that is the end of the remaining missionaries- the cannibals will finish them off.

(Refer Slide Time 18:09)



The question is, how do we transport them to the other side of the river? Let me first show you 1 solution for doing this. In this solution, let us see how we transport them to the other side. Let us say that I will use rounds for the missionaries and crosses for the cannibals. This is the initial state, where all the missionaries and all the cannibals are on the left side of the bank. This is our boat, right? So, this state I will denote by 3, 3 and 0, where this 3 denotes the number of missionaries, this 3 denotes the number of cannibals and this flag, which will toggle between 0 and 1, indicates that the boat is now in the left bank, right? Okay.

Then, what are the things that we can do? We can transport 1 cannibal and 1 missionary to the other side, or we can send 1 cannibal, or we can send 1 missionary. If we send 1 missionary, then, the number of cannibals will have a majority on this bank, because we will have 3 cannibals and 2 missionaries left. So, that will be the end of these 2 missionaries, right? Let me first show you the solution, then, we will see what is the formulation of the problem. Let us say, that I start by sending 2 cannibals to the other side. If I send 2 cannibals to the other side, then what I have here is 3 missionaries here, and 2 cannibals here, 1 cannibal here. But the boat is now on this side. This is the state where I have 3 missionaries here, 1 cannibal here and the boat is on the other side. So, 1, right? Then next, 1 of the cannibals will have to bring the boat back. So, this cannibal will take the boat back.

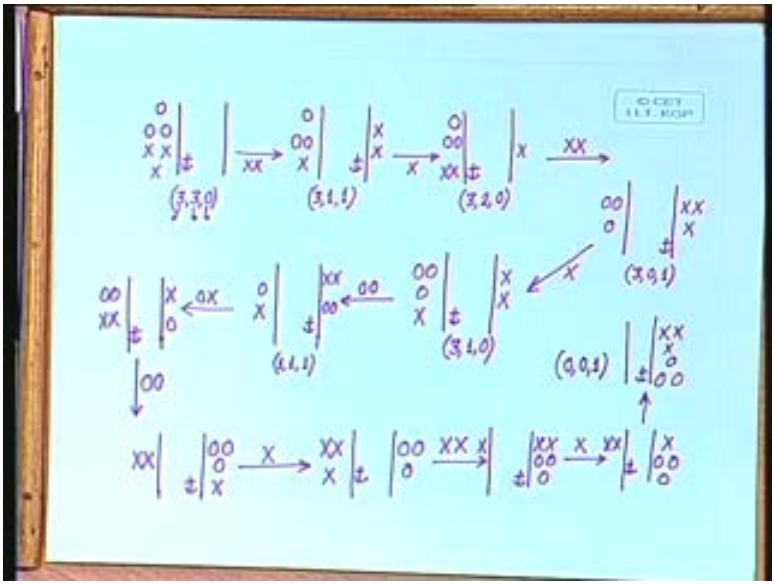
In the next state, I will have 3 missionaries here, 2 cannibals here and 1 cannibal on the other side. The boat is here and this state is 3, 2 and 0, right? Then, what do I do? Okay. Next, I will send 2 cannibals to the other side. If you send 1 missionary and 1 cannibal, you will land up in trouble in the next step, so that is where the search paradigm will come in. I know the solution. That is why I am writing it down cleanly, but the search algorithm will attempt to find out in an automated way. The next After these 2 cannibals

are on the other side, I will have all 3 cannibals on that side and all 3 missionaries on this side, and the boat is on the other side, right? This is the state where I have 3, 0 and 1.

In the next state, 1 of the cannibals will bring the boat back and the state that I will have is 2 cannibals on that side, 3 missionaries here and 1 cannibal. And the boat is here. So, this is the state- 3, 1 and 0, right? Then, I will send 2 missionaries to the other side. If I send 2 missionaries to the other side, so I will have 2 cannibals and 2 missionaries on the other side, and 1 cannibal and 1 missionary here. And the boat is on the other side. So, this is one, 1 and one. Then, I will send 1 missionary and 1 cannibal to bring the boat back, right? If I send 2- if I send 1 cannibal, then I will be in trouble on this bank. If I send 1 missionary, then the remaining missionaries will be in trouble in this bank. So, I have to send 1 missionary and 1 cannibal. When I send 1 missionary and 1 cannibal back here, then what I will have here is 1 missionary, 1 cannibal here 2; missionaries and 2 cannibals here, and the boat is now in this side, right? Then, I will send both the missionaries to the other bank. So, I will have 2 cannibals here, all 3 missionaries here and 1 cannibal here. And this, right? Then the remaining is part is easy, right? You send the cannibal back. You have all 3 cannibals here, all 3 missionaries here, and the cannibals have the boat.

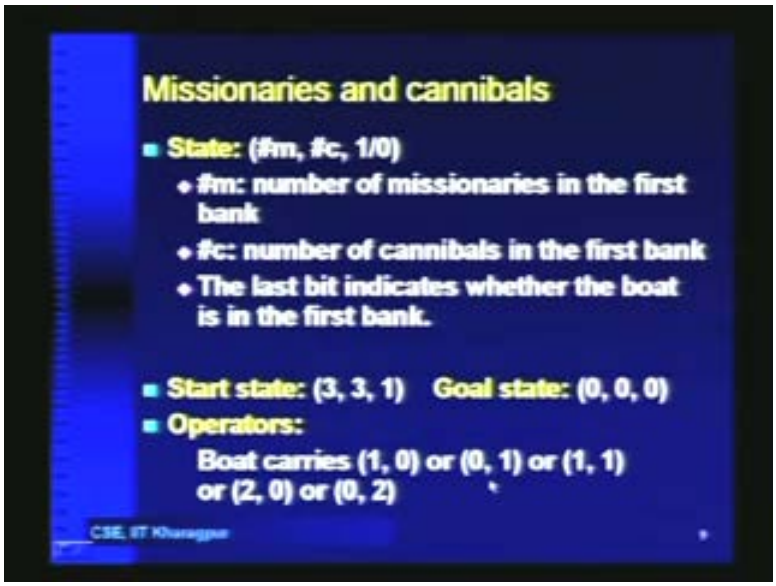
So, now they can transport themselves. What they will do is, in 2 steps- first step: 2 will go to the other side, right? I will have 1 cannibal here and 2 cannibals, 3 missionaries, here. Boat is on this side. Then, 1 cannibal will come back. So, I will have 2 cannibals here, 1 cannibal and 3 missionaries here. And the boat is now on this side. And then, finally- these 2 cannibals will go over to the other side. I will have 3 cannibals here, 3 missionaries here and boat is here. The final configuration that I wanted to reach was 0, 0 and one, right?

(Refer Slide Time 24:53)



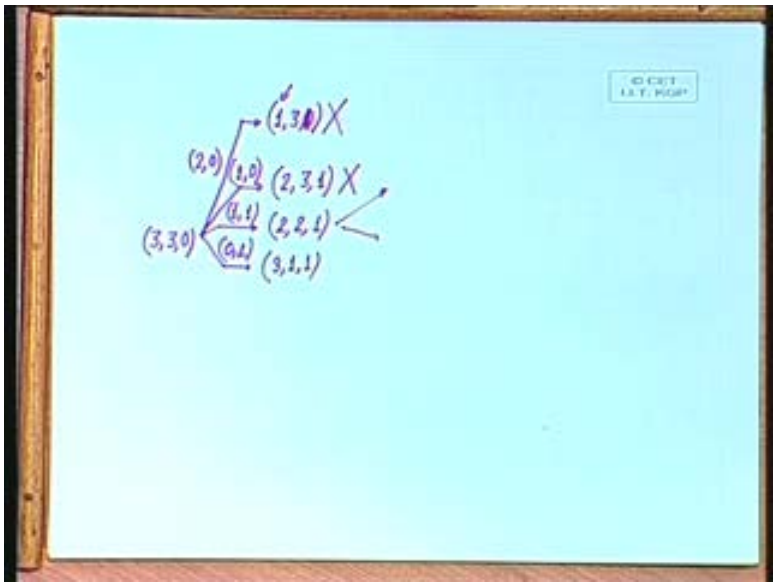
Now, we know what will be the formulation of the problem. I will have the state as #m, #c, which means the number of missionaries here, the number of cannibals- and the last bit indicates whether the boat is in the first bank or in the second bank- and the start state is 3 3 actually 3, 3, 0. And the goal state is 0, 0, 1. Depending on how you interpret the flag. And the operators are: the boat carries 1 missionary, 0 cannibals; or 0 missionaries, 1 cannibal; or 1 missionary, 1 cannibal; 2 missionaries, 0 cannibals; and 0 missionaries, 2 cannibals. The nice way of searching this would be to start from the start state and systematically apply these operators, one by one, and see whether you are able to reach the goal.

(Refer Slide Time 26:06)



You will start from one of the states and then see what happens. Suppose we start from the state 3, 3, 0, and then I take 2 missionaries to the other side. That will leave me in a state: 1, 3, 0; 1, 3, 1, rather, and this state is a state where it is the end of this missionary. So, we can backtrack from here and try instead, sending 1 missionary and 0 cannibals. If we do that, then we will have 2, 3, 1, which is again a bad state, because these 2 missionaries will go. Then, you try applying the other operator. You send 1, 1; then that will leave you in 2, 2, 1- which is fine- and then, again, from this, you can try some other operators or you can try some other operators here. For example, you can try 0, 2, to get 3, 1, 1- right? This is how the search tree is going to grow, but if you look at it, the search space can be quite large. But, it is not just a tree, it is a graph, because in various sequences of operators can actually take you to the same state.

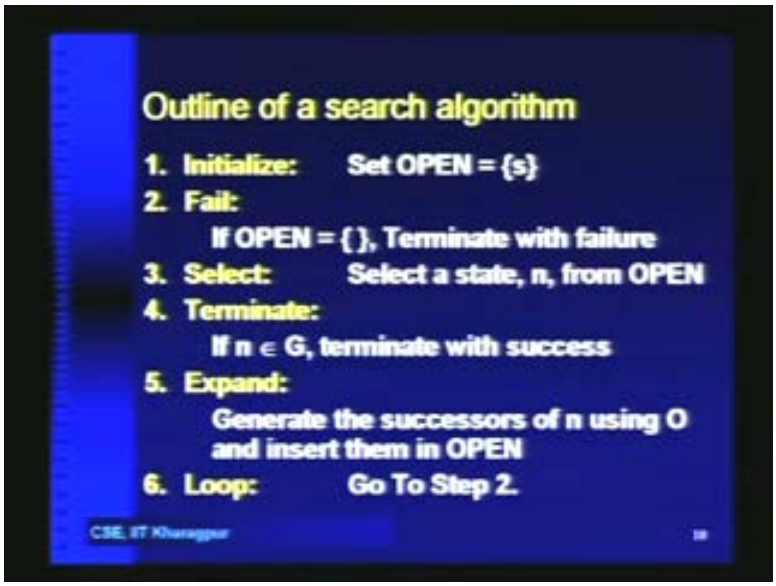
(Refer Slide Time 27:39)



So, there will be necessity to detect whether there is a state which is revisited, otherwise, if you apply something like depth first search, and in DFS, if you cannot determine whether you have visited some state, then you can potentially go into an infinite loop. So, we will have to devise mechanisms by which we are able to check whether we are visiting some state or we are arriving at the same state, along some other path, right? And so on. Now we will look at the outline of our first basic search algorithm and we will develop this search algorithm to encompass all the different kinds of AI search algorithms that we will study. We will start by maintaining a list called OPEN. This is a-, this terminology of a list called OPEN was devised many decades back, and so we will continue with the same names and jargon. Initially our list OPEN will contain the start state- this is the initialization. Then, we have 1 termination criterion. If we find that the list OPEN is empty, then we will terminate with failure, right? Let us look at a little bit more, then it will become clear. Third step is, we select a node n from OPEN. We select any state n from OPEN. And then we check whether n is a goal. If n is a goal, then we have found a goal, so we terminate with success. Remember, there, our objective is to start from the start state and find a path to a goal state. So if n belongs to G, then we terminate with success. Otherwise we generate the successors of n using the transition operators O, and insert these new states in OPEN, right?

So, what we are doing is, we are starting from 1 state and then expanding that state to generate its successor states by using the state transition operators, and inserting these states into OPEN. At this point of time, I have not mentioned whether OPEN is a FIFO or LIFO, or whether it is a priority queue. Nothing. And these algorithms will vary depending on the kind of data structure that we use for OPEN. And then finally, we go to step 2. After expanding and inserting the children in OPEN, we go back to step 2. Now think of this algorithm- just study this algorithm. Slide?

(Refer Slide Time 31:07)



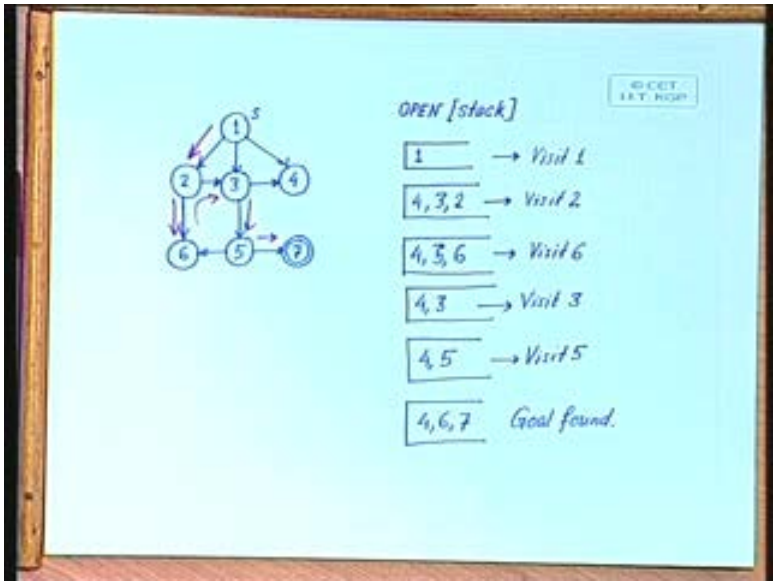
Look at this algorithm and try to figure out what kind of algorithm this will be, if OPEN is a queue and if OPEN is a stack. (Student asks question.) No, if you treat OPEN as a stack, then you will select the top element from OPEN. If the OPEN is queue, then you will select the last element, right? Let me give an example. (Student speaking). Right. So, if the so if OPEN is a queue, then we will have breadth first search, and if OPEN is a stack, then we will have depth first search. See, think of the recursive depth first search. Where is the stack? It is the recursion stack. You have visited from one, pushed that node, because you have not yet visited all its successors, right, and picked up one successor, gone down to that successor and again pushed that node, because you have not yet seen the other successors. So, when you backtrack from a node, you go back to the previous parent and pop it up and then start visiting its other successors, right?

Let us see this through an example. The graph that we have is, say- this is state 1- and then let us say, suppose this is the start state, s. Initially, let us see the contents of OPEN, and here we will treat OPEN as a stack. Now, in the first step, what do we have in OPEN- we have 1. In the first step, what we are going to do is- we are going to select. If you look at the slide, the first step is to select a state n. In step 3, select a state n from OPEN and then if it is not a goal, then expand and generate its successors. Coming back to here, if we expand this, then what will happen is- here we visit 1 and then generate its successors, and let us say we put them in this order- 4, 3, 2, right? In the next step, we are going to pop out 2 and visit 2. When we visit 2, let us say that we find that its successors are 6 and 3. So, what happens is, 2 comes out and 3 is already there. We just put 4 followed by 3 is there, and we have 6.

In the next step, we visit 6. Let us say 6 does not have any next state. So, we will backtrack. The next thing that is going to pop out is 3, so we will have a state where we have 4, 3. And now we will visit 3. When we expand 3, we find that it has 2 successors- namely 4 and 5. The OPEN contents 4 and 5. Next, we visit 5, right, and we find that 5

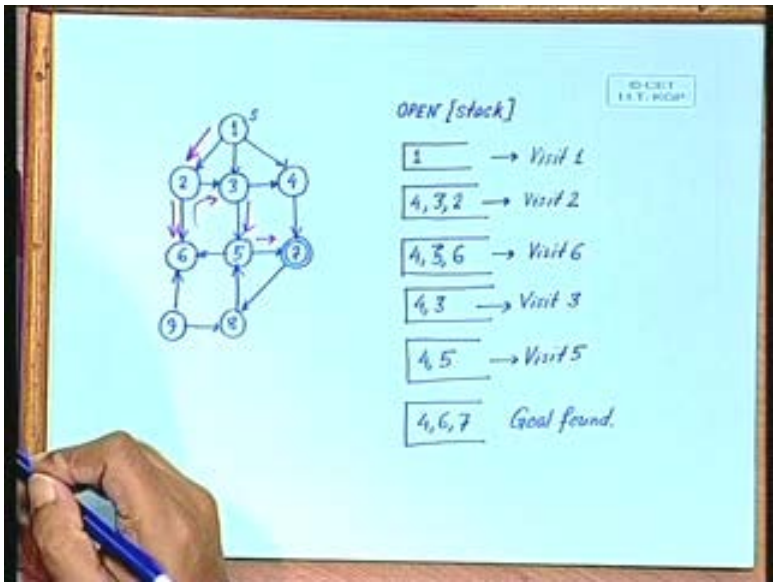
has successors 6 and 7, which happens to be a goal node. So, what happens is, when we visit 5, we will have 4 and 6 and 7, and as soon as we have visited a goal, then we stop. So, we have goal found. If you look at the sequence of states visited, then you see we have gone from 1 to 2, then 2 to 6, then backtracked and then went to 3, and 3 to 5 and then 5 to 7. And this is exactly what we do in DFS, right?

(Refer Slide Time 36:53)



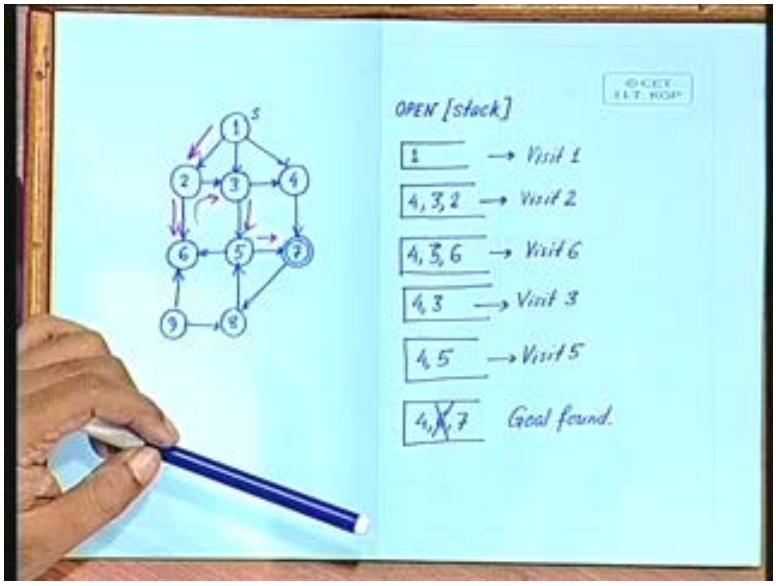
Okay. In your state space, you can also have other transitions, like, you can have a state 8, which are not visited at all by the search algorithm. And you can also have some states which are not reachable. For example, if you look at the state 9, it is not reachable from the start state. But the whole point is that, at no point of time do we want to make the whole state space explicit. We want to only unfold that portion of the state space which is necessary to find out the goal. So, nodes like 8 and 9 never came into the picture here, because those were not necessary for this depth first “ “

(Refer Slide Time 37:45)



Now, if instead of using a stack, we used a queue, how would it have looked? Yes. This one. Yes. (Student speaking). No, but I did not visit 6. Oh. Here. Yesyes. This 6 should not come, right? This 6 should not be entered in OPEN again, because we have already visited, but I have not yet told you how we maintain this information about what we visited.

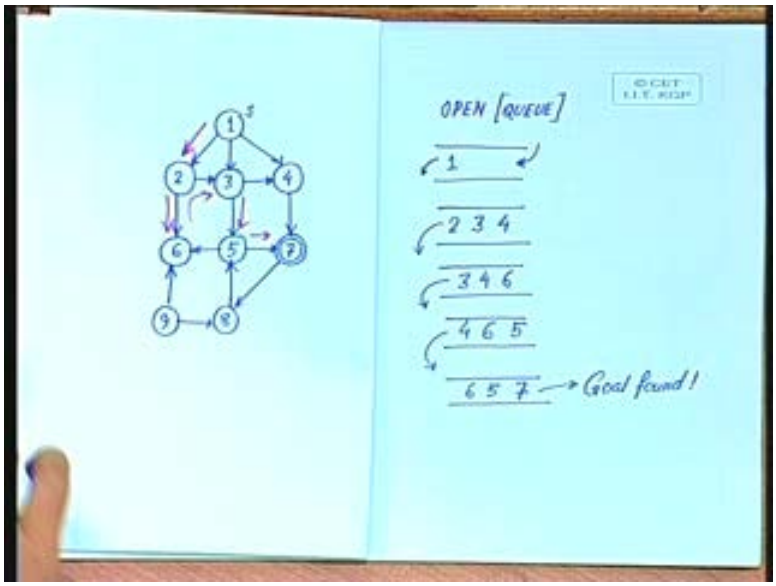
(Refer Slide Time 38:37)



Later, I will show you that we will maintain another list called CLOSED, which will keep track of those states which we have already visited, but before that just let us quickly see what happens if we do a BFS with this same graph. In that case, what we will do is-

OPEN will be treated as a queue. OPEN will be treated as a queue, so initially we will have 1, right? I will enter nodes through this side and take them out through this side, as in a FIFO. So, when I visit 1, then I will enter the successors- 2, 3, 4, right? Then, I will take out 2 and when I take out 2, then, I will insert its successors. 3 is already there, so I will not reinsert it. I will insert 6. That is right. Next, I will be taking out 3. This is where we deviate from the DFS. So, will be taking out 3 and when I take out 3, I will insert 5, right? Next, I will take out 4 and I have 6, 5 and I get 7. And since this is the goal, so I will say that as I have found the goal, right? This is breadth first search. So, if I treat OPEN as a stack, then I have DFS.

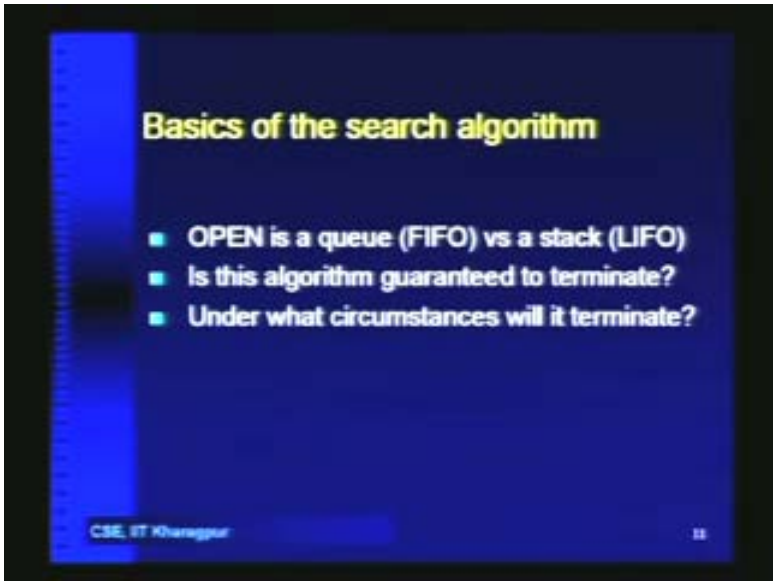
(Refer Slide Time 40:40)



If I treat OPEN as a queue, then I have BFS, but there is a significant difference between the 2, in terms of the kind of space that I am going to require. So, let us see the following things. Suppose we have used OPEN as a FIFO. Is the algorithm guaranteed to terminate? In breadth first search? But you must remember one thing: is that when we are talking about the state spaces, we are not necessarily talking about finite state spaces. The state spaces can be infinite. For example, if you are working on state transition operators which work on real numbers or integers, then potentially you can go from one state to another. And because you have integer functions giving you the next state, you can continue forever, right?

So, in breadth first search, we are looking at breadth first. That is why, if you have a goal in a finite depth, you will terminate. But in depth first search, if the tree is infinite, then there is a possibility that you go down an infinite path, and the goal is on some other path. So, depending on the size of the state space, if your size of the state space is very large in depth, then if you do depth first search, then you might actually end up doing a lot more work. This analysis is what we will do later on, okay? And then under what circumstances will it terminate? Again, we will look at this problem in more detail subsequently.

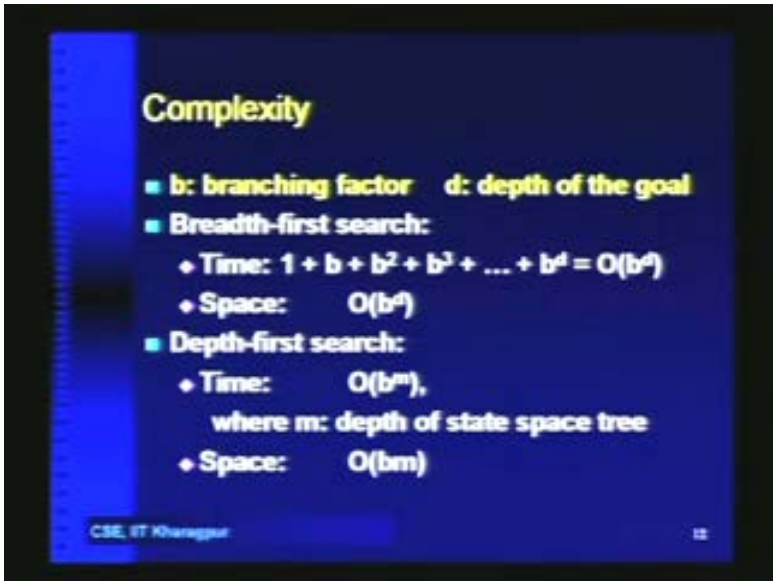
(Refer Slide Time 42:38)



Let us look at the complexity of the breadth first search. Suppose b is the branching factor. By branching factor, I mean that the number of transition operators that you have. So, given a state, that is the number of different next states that you can possibly have. And d is the depth of the goal. If you do breadth first search, then on an average, this is the time that you are going to require. One, first the start state. b for the set of states at depth 1. b square states at depth 2, b cube depth states at depth 3, and so on. And you will visit, in the worst case, all states up to depth d. So, in the last iteration, you will have b to the power of d states, in the d^{th} level, right?

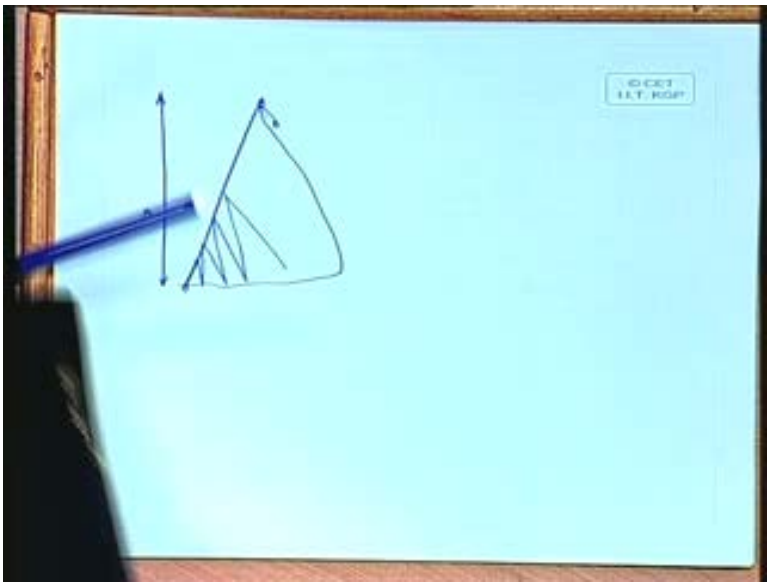
If you take the sum of these, it comes to order of b to the power of d. And the space required is also order of b to the power of d, because since you are doing breadth first search, when you are starting with a node at depth d, you must be having all the other states at depth d already in OPEN, right? The space requirement is order of b to the power of d. This is what kills breadth first search. Otherwise, this is a nice algorithm, but the space requirement is a more stringent requirement in the kind of problems that we are working at, because after some time, you will be eating up so much of memory that your system will start crashing. We will do some assignments to actually test the limits of the system, and you will see that if we start by simple things like 15-puzzle, that is enough to kill the kind of desktop machines that you have. That is enough. Okay. We will see those things later.

(Refer Slide Time 44:52)



If we look at the complexity of depth first search, then the time required is order of b to the power of m, where m is the depth of the state space tree. Now, note that in depth first search, the depth of the goal is not important because you are not going level by level. Rather, what you are doing is, you are going from straight down as far as you do, and if m is the depth of your state space, then you will do right down here, and then start backtracking and trying out the other parts, right? This is how depth first search will work. So, even if your goal is here, you know you will end up visiting all these parts and then come back and find that the goal is here. That is what happens in depth first search. If you look at the complexity, it is order of b to the power of m, where m is the depth of the state space tree.

(Refer Slide Time 45:56)

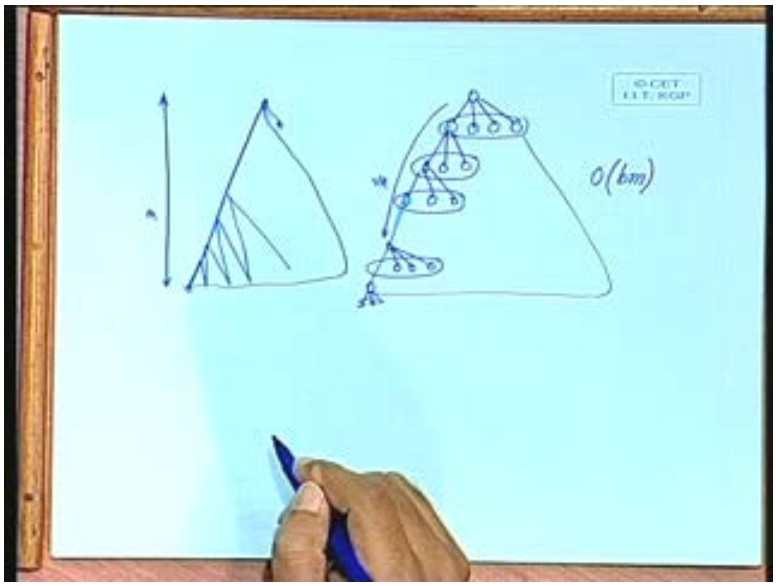


Is this clear? Yes? Oh, the state space tree is the tree that you obtain by applying the state transition operators repeatedly on the set of states. So, you start with the start state. These are the next states that you can generate with your state transition operators. From this state, these are the states that you can generate with your next state operators, right? If you keep on unfolding this, what you get is the state space tree, right. And where do I end this state space tree? When I find that all the next states of this state are already there in the state space tree. They are already there in this path. You have already visited that. That is where there are no new states, so you backtrack and try another one, or you reach a state from where there is no state transition operator.

So, if you have a state space tree of depth m, then the time required is order of b to the power of m, because it is going to go right down, up to depth m, and then in the worst case, visit all nodes there and come back to the top, and find that the last node in the top of the start state- the last child of the start state is the goal state. That is the worst case scenario, and there you will require order of b to the power of m. But the good thing is that the space complexity of depth first search is order of bm. Why bm? I am going down one path, right? If you just look at the path, what I am pushing in OPEN is- in the first round. I will push this in OPEN. In the second round, I will push this in OPEN. In the third round, I will push this in OPEN. In the fourth round, I will push this in OPEN.

At every round, I am pushing the node that I am visiting and the brothers of that. So, if I have if the depth of this tree is m, then I have m nodes here, and each m having b brothers, right? So, my OPEN is not going to be larger than order of bm.

(Refer Slide Time 48:55)



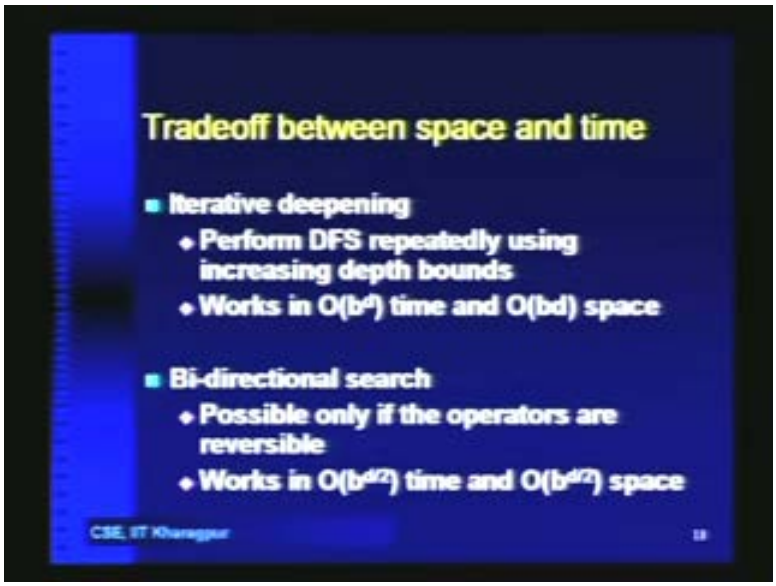
Now, if you compare that with the complexity of breadth first search, then you see that this is significantly better. For breadth first search, we had order of b to the power of d, and here the space complexity is order of bm. And typically, bm can be much lesser than order of b to the power of d. So, that is why depth first search is useful for some kinds of algorithm; many scenarios, right? Now, there are some nice trade offs that we can have between space and time. We saw that in breadth first search, we were able to do it quickly, whenever the goal is near the start state. And the drawback was, it was using up too much of space. So, one nice trade-off between space and time is, we will use depth first search to do breadth first search. What we are going to do is what is called an iterative deepening. We will perform DFS, but by repeatedly using increasing depth bounds. Let us see how we go about doing this.

We will start from the start state and do a DFS of depth 1. We have seen all the steps at depth 1, but we have done it with DFS, mind it. If we have found the goal, we terminate. Otherwise, we go to depth 2. Again, do a DFS up to depth 2. If you have not found the goal, do a DFS up to depth 3. We continue in this way until we have found the goal. If you look at the complexity now, we are working in order of b to the power of d time, and order bd space. The space is easy to see, right, because in the first iteration my space is b. In the second iteration my space requirement is 2b. In the third iteration my space requirement is 3b, and the d^{th} iteration, I will get the goal. If d is at If the goal is at a depth d, then in the d^{th} iteration, I will have the goal.

So, order bd is the worst case space that I will require, but that it works in order b to the power of d time, is not so obvious, right? In the first iteration, it is going to take order of b, then order of b square, then order of b cube; unlike space, where you will take the max of the space requirements in the different iterations. For the time, they will add up- for the time complexities, they will add up. In the first iteration, you had order of b, second iteration: order of b square, third iteration: order of b cube. So, b plus b square plus b

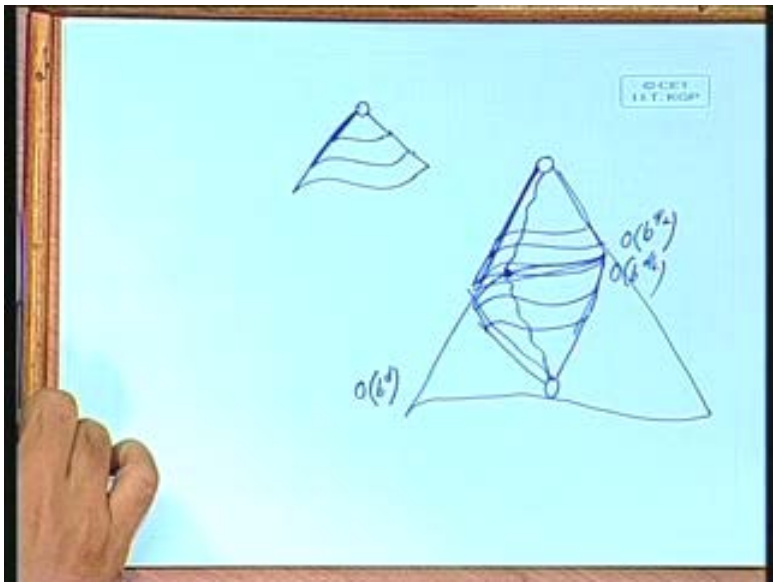
cube plus… up to b to the power of d. That is also going to come to order of b to the power of d. So, please check out this recurrence. One final thing is bi-directional search, where what you can do is, you can do breadth first search, but from 2 directions. You can start from the start state, work in 1 direction and start from the goal state.

(Refer Slide Time 52:48)



I start from the start state and proceed like this; start from the goal state, proceed like this and then I do 1 iteration of breadth first search for this. And we keep on doing this until, at some point of time, there is some common node which I find between the 2. When I found that, then I know a path from here to here, I know a part from here to here. So, I have found a path to the goal. This helps because instead of having done this, which would have taken order of b to the power of d, I am not doing- in the worst case- this part and this part. This is order of b to the power of d by 2, and again, this part is order of b to the power of d by 2, right? So, the total is again order of b to the power of d by 2.

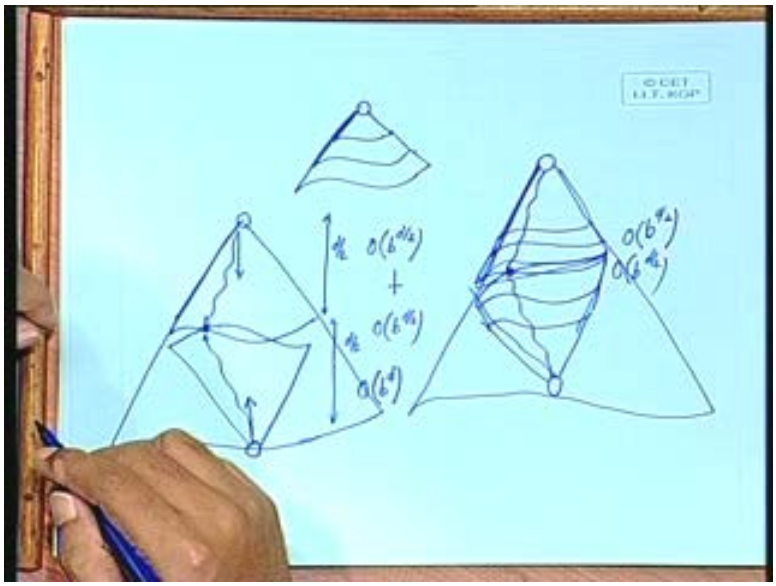
(Refer Slide Time 53:52)



It works in order of b to the power of d by 2 time and order of b to the power of d by 2 space, but this can only work if the state transition operators are reversible, so that you can start from the goal and work backwards. And in puzzles like 15-puzzle, etc. it is indeed reversible. You can try out from both sides. (Student asks question.) Of the bi-directional search? Well, you can start in, if you did normal breadth first search, then you will go straight from the start state and expand all nodes up to depth d. So, your complexity will be order of b to the power of d. Here, what we are doing is, we are starting from the start state and moving in this direction, and we are also starting from the goal state and moving in the other direction, using the reverse of the state transition operators. And we continue until the frontiers of the searches from these 2 sides meet at some point of time at some place.

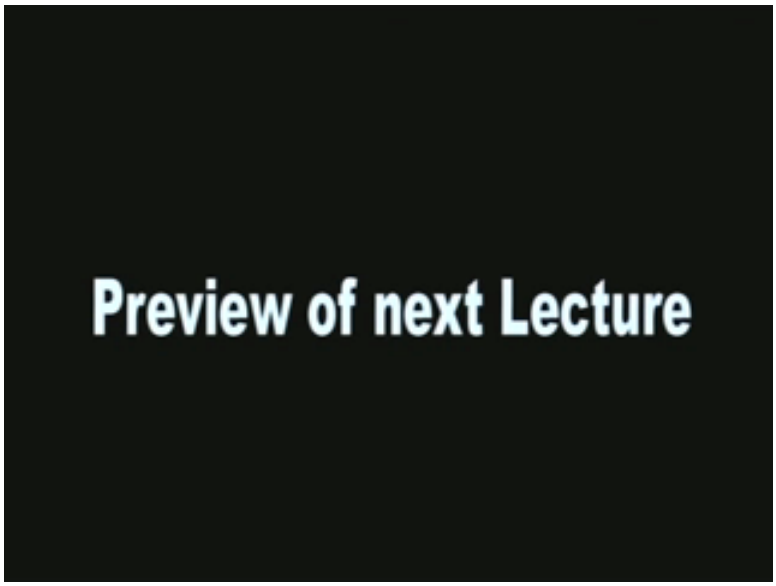
When you have a common node in these 2, then you can find out a path from the start state to the goal state. Now you see that because you were doing 1 iteration from this side, 1 iteration from that side- so, you will be doing d to the power of 2 iterations from this side and d to the power of 2 iterations from the other side, right? Order of d to the power of 2 from this side, order of d to the power 2 from the other side. If you do d to the power of 2 iterations from this side, your complexity for BFS is order of b to the power of d by 2; and also from the other side, it is order of b to the power of d by 2. If you add up this plus this, you still have order of b to the power of d by 2, right?

(Refer Slide Time 55:43)



So, you have actually reduced the complexity by half, right? If you have order of b to the power of d, you have order of b to the power of d by 2, which is significant actually, right? We will conclude this lecture here and from tomorrow onwards, we will start looking at the search problem with the introduction of costs. We will see what happens when the state transition operators have associated cost when the goals have associated costs.

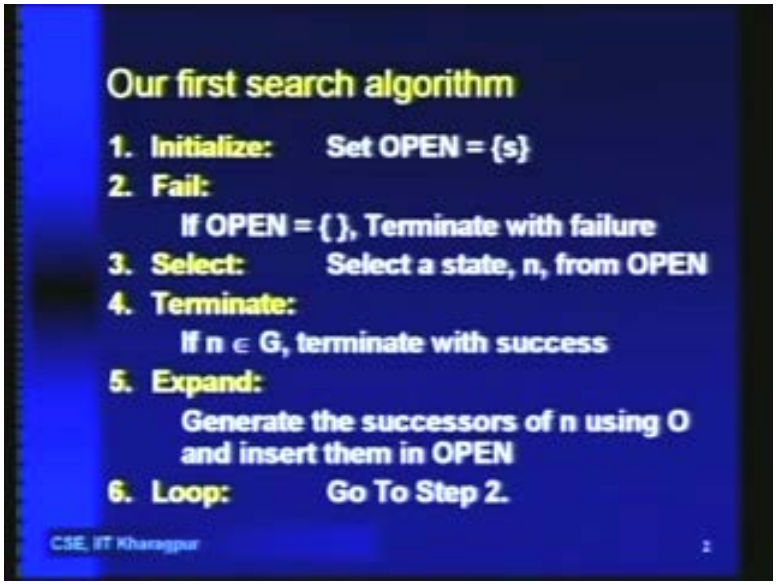
(Refer Slide Time 56:28)



Today, we will start with the next chapter on searching with costs. In the last class, what we had done was, we studied the first search algorithm which was as follows: that we

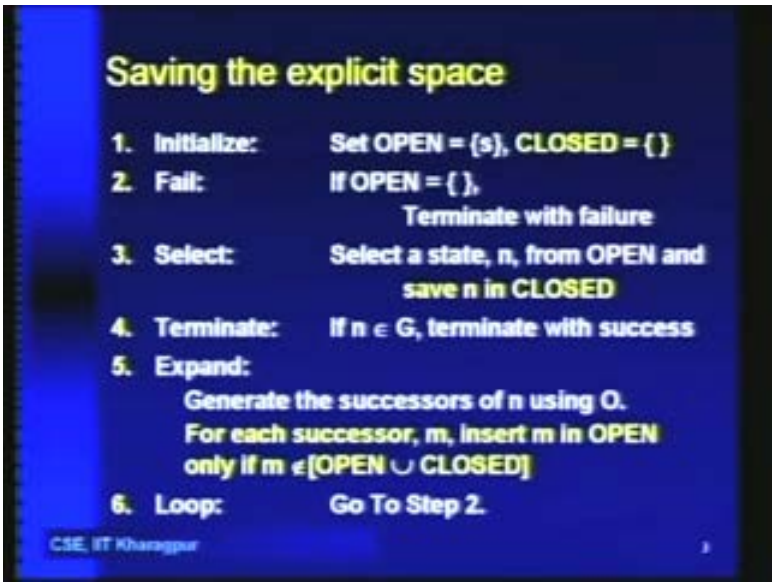
start with initialize and we were maintaining a list called OPEN. We put the start node in s. Then, if we find at some point of time, that OPEN is empty and we have still not found the goal, then we terminate with failure. Then select a state n from OPEN. Terminate: if n is a goal- then we terminate with success. Otherwise, we generate the successors of n using O and insert them in OPEN, and finally go to step 2. So, what we are essentially doing is, we are progressively maintaining a frontier of the nodes. And in each iteration, we are picking up a node from the frontier and expanding that. That extends the frontier and we continue in this way.

(Refer Slide Time 57:48)



OPEN maintains the current frontier. If at any point of time, we find that OPEN is empty, that means we have reached the end of the frontier and we have still not found the goal. So then, we will terminate with failure. One thing that I had not mentioned so far is that- how do we maintain the part of the state space that we have already visited? And that is important, because there are cases where we have to know that node is already visited so that we do not revisit that node. This is the extension of the same algorithm, where we will save the explicit state space. Let us see how we do that.

(Refer Slide Time 58:34)



See, this just shows the difference with the previous algorithm. The only changes that I have made are highlighted in yellow. We are now maintaining another list called CLOSED. When we select a state n from OPEN, here we save the state in CLOSED, right?