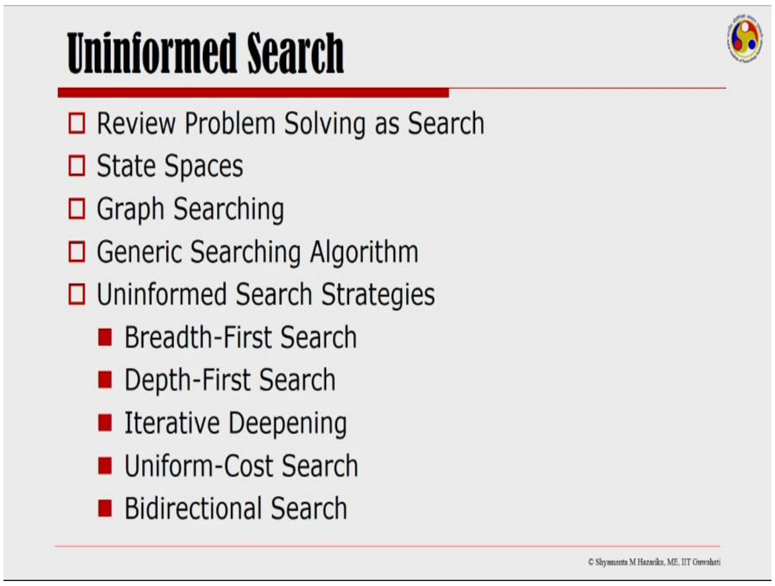
Fundamentals of Artificial Intelligence

Prof. Shyamanta M Hazarika
Department of Mechanical Engineering
Indian Institute of Technology - Guwahati

Module - 1
Lecture - 3
Uniformed Search

Welcome to fundamentals of artificial intelligence. Today we are going to look at uninformed search. Usually, in most of the AI techniques that we will explore, we will later look at using domain information or what is more commonly called heuristic knowledge which would be under the category of informed search. However, as an introduction to search in graphs, we would love to look at uninformed search first. So, what we will cover today is:

(Refer Slide Time: 01:06)



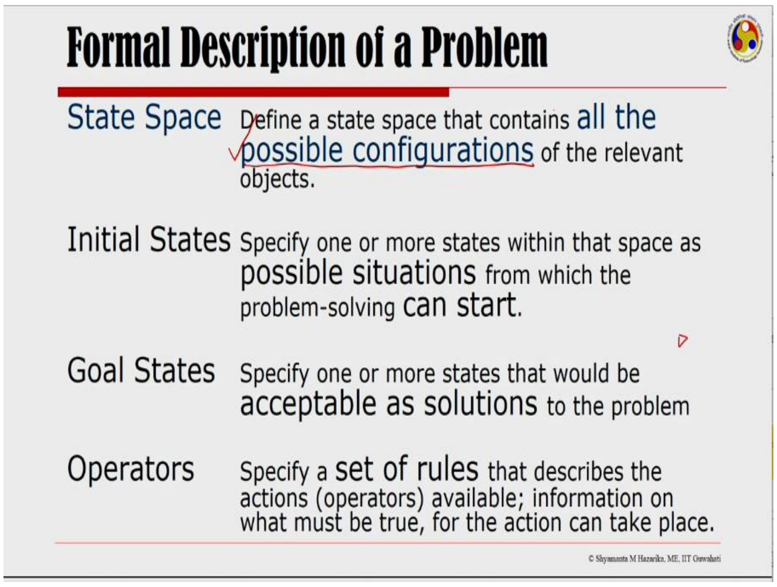
Uninformed Search

- Review Problem Solving as Search
- State Spaces
- Graph Searching
- Generic Searching Algorithm
- Uninformed Search Strategies
  - Breadth-First Search
  - Depth-First Search
  - Iterative Deepening
  - Uniform-Cost Search
  - Bidirectional Search

© Shyamanta M Hazarika, ME, IIT Guwahati

We will very quickly review problem solving as search; look at what we meant by state spaces. We would then formally introduce graph searching and introduce a generic searching algorithm. Thereafter, we would look at a couple of uninformed search strategies. First, the breadth-first search, thereafter the depth-first search, iterative deepening, uniform-cost search; and then we will look at something called bidirectional search.

(Refer Slide Time: 01:40)



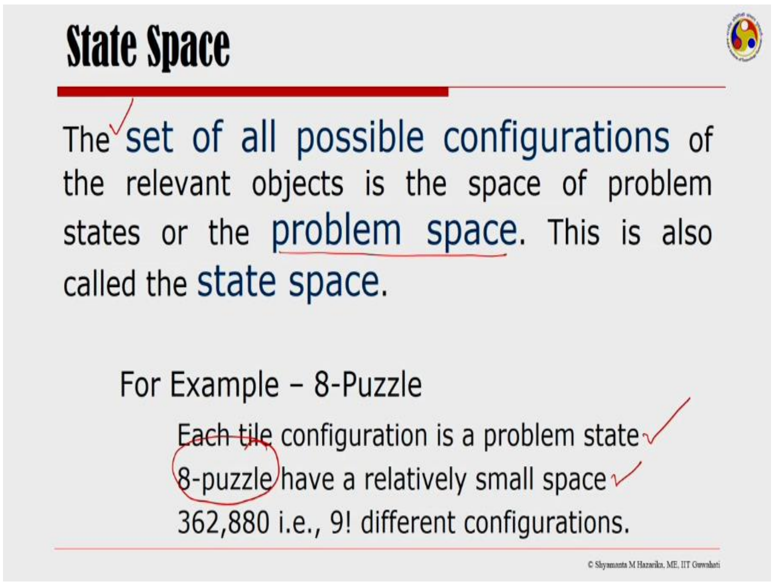
Formal Description of a Problem

State Space	Define a state space that contains all the possible configurations of the relevant objects.
Initial States	Specify one or more states within that space as possible situations from which the problem-solving can start.
Goal States	Specify one or more states that would be acceptable as solutions to the problem
Operators	Specify a set of rules that describes the actions (operators) available; information on what must be true, for the action can take place.

© Shyamanta M Hazarika, ME, IIT Guwahati

Let us look at what we meant by problem solving as search. We define a state space that contains all the possible configurations. 1 or 2 configurations I have shown you. We would love to define all possible configurations of the relevant objects. We would then look at the initial state. That is, specify 1 or more states within that state space as situations from which we start. And then, we define something called the goal state; 1 or more states that would be acceptable as solutions to the problem. And of course, we need to specify the operators; a set of rules that describes the actions or operations available.

(Refer Slide Time: 02:26)



State Space

The set of all possible configurations of the relevant objects is the space of problem states or the problem space. This is also called the state space.

For Example – 8-Puzzle

Each tile configuration is a problem state

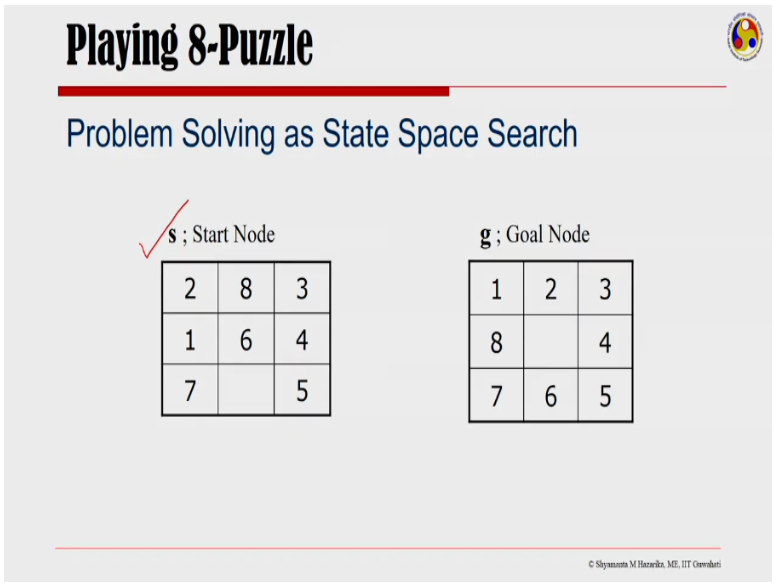
8-puzzle have a relatively small space

362,880 i.e., 9! different configurations.

© Shyamanta M Hazarika, ME, IIT Guwahati

The state space is what is vital for formulating a problem solving domain as state space search. The state space literally consists of all possible configurations. This is also referred to as the problem space, for the 8-puzzle game that I have shown you little while ago. Each tile configuration is a problem state. The 8-puzzle, we should note has a relatively small space. Okay. There could be problems which would have huge state space. And we would look at how to search it in order to get to a solution.

(Refer Slide Time: 03:10)



Playing 8-Puzzle

Problem Solving as State Space Search

s ; Start Node

2	8	3
1	6	4
7		5

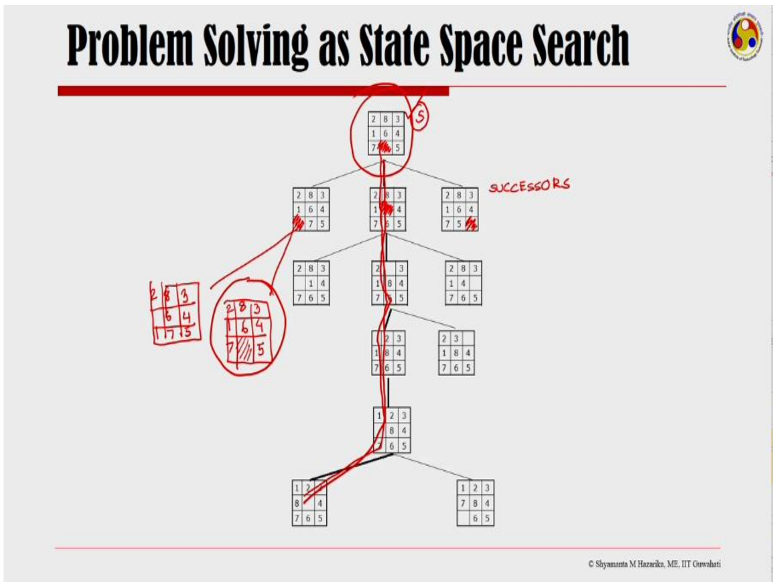
g ; Goal Node

1	2	3
8		4
7	6	5

So, let us look at a problem solving as state space search. And this is the start node which I wanted all of you to take note of. This is as with 9 tiles here. One of them is blank and you move the blank around to arrive at this goal position. Now, in the last class, we have looked at how these expansions can take place given the 4 operations.

(Refer Slide Time: 03:38)

Problem Solving as State Space Search



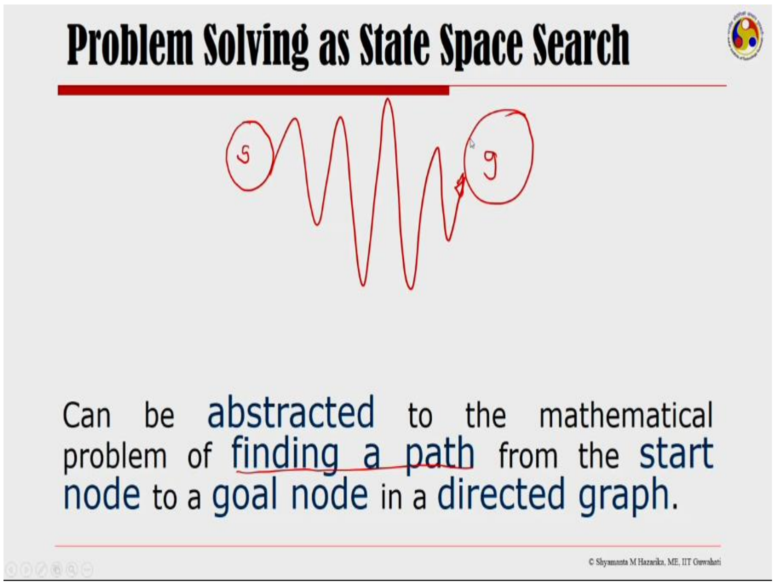
We will just look at the complete tree of solution. So, given this state, you could have 3 possibilities of moving the blank. So, the blank is here. You could think of moving the empty to the left. It could come here at this point. Or you could think of moving it up to the center or you could move it to the right. So, given the start S, this sets, these configurations of tiles are the, its successor nodes, successors.

And we can keep on generating the successors. Like, from here, we could get at more successors by moving the blank to a newer configuration. Like the blank would go up and one could come down. So, we could have 2, blank, 1, 8, 6, 7 and 3, 4, 5. And we could have another one and which will be like 2, 8, 3, 1, 6, 4; 7 could go this side. And I would have blank here. And I would have 5.

So, we could generate. I had all the successors from 1 node. But one thing to note here is that this point here, this node here is a repetition of what I started with here or as my start. So, I would avoid it getting expansions of these type of successors. As having said that, I would love to get all the successor nodes listed. And then, the idea would be to look for a path that takes me from, through this a space of successors as to the configuration that I was looking for.

And therefore, this is line of expansions would be the solution to the problem that I was looking for. Now, if you look at this very closely, we can have a simile here. This can be thought of as a graph.

(Refer Slide Time: 05:48)



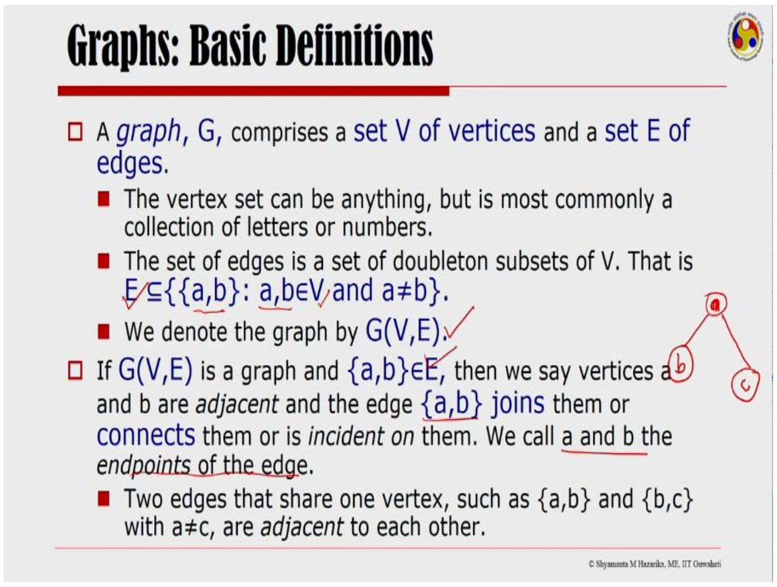
Problem Solving as State Space Search

Can be abstracted to the mathematical problem of finding a path from the start node to a goal node in a directed graph.

© Shyamanta M Hazarika, ME, IIT Guwahati

And basically, what were we looking for is given a node, start node of a graph and a goal node g. I am looking for some path that takes me from s to g. So, problem solving as state space search can be extracted to the mathematical problem of finding a path from the start node to the goal node or in a directed graph.

(Refer Slide Time: 06:22)



Graphs: Basic Definitions

- A *graph*, G, comprises a set V of vertices and a set E of edges.
  - The vertex set can be anything, but is most commonly a collection of letters or numbers.
  - The set of edges is a set of doubleton subsets of V. That is $E \subseteq \{\{a,b\}: a,b \in V \text{ and } a \neq b\}$.
  - We denote the graph by G(V,E).
- If G(V,E) is a graph and $\{a,b\} \in E$, then we say vertices a and b are *adjacent* and the edge {a,b} joins them or connects them or is *incident on* them. We call a and b the *endpoints of the edge*.
  - Two edges that share one vertex, such as {a,b} and {b,c} with $a \neq c$, are *adjacent* to each other.

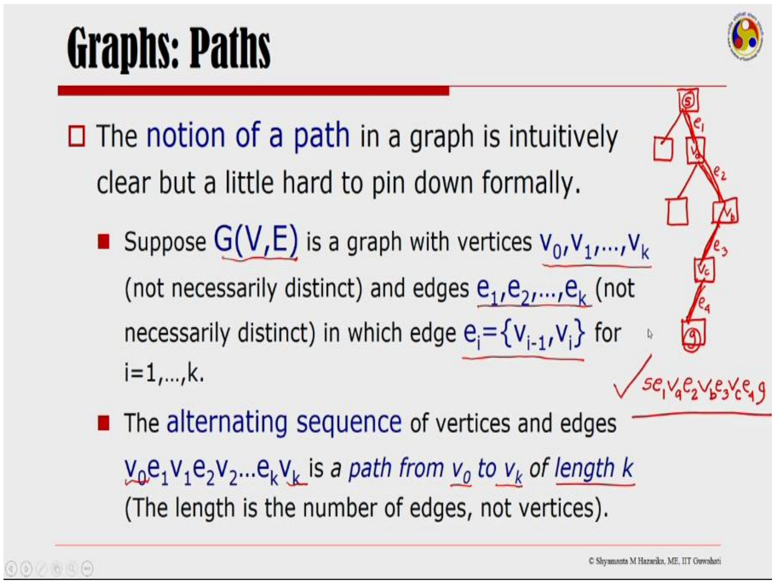
© Shyamanta M Hazarika, ME, IIT Guwahati

So, let us take a look at what we mean by graphs and directed graphs before we proceed further into our discussion of problem solving as state space graph. You must have looked at graphs before, but for completeness, let me put the definition here. A graph G comprises a set V of vertices and a set E of edges. The vertices as can be anything but is most commonly a collection of letters or numbers. That's the representation.

And the set of edges is a set of doubleton subsets of V. That is, E is as a group of a subset of a, b; where a, b belongs to the vertices and a is not equal to b. U sually, the graph is denoted as a 2 tuple G V E. So, if a graph is given and I know its edges, then we say vertices a and b are adjacent. And if there is an edge a b that joins them in the graph. And we call a and b themselves as the end points of the edge.

2 edges that share a vertex; such as, let us say I have a graph here and have a here and b here. And that's, that's 1 edge. I have another edge a and c here. That's another edge. So, if I have a b and a c, here they are sharing a common node a. Then they are said to be adjacent to each other.

(Refer Slide Time: 08:03)



Graphs: Paths

- The notion of a path in a graph is intuitively clear but a little hard to pin down formally.
  - Suppose $G(V,E)$ is a graph with vertices $v_0, v_1, \ldots, v_k$ (not necessarily distinct) and edges $e_1, e_2, \ldots, e_k$ (not necessarily distinct) in which edge $e_i = \{v_{i-1}, v_i\}$ for $i = 1, \ldots, k$.
  - The alternating sequence of vertices and edges $v_0 e_1 v_1 e_2 v_2 \ldots e_k v_k$ is *a path from v_0 to v_k of length k* (The length is the number of edges, not vertices).

✓ $s e_1 v_a e_2 v_b e_3 v_c e_4 g$

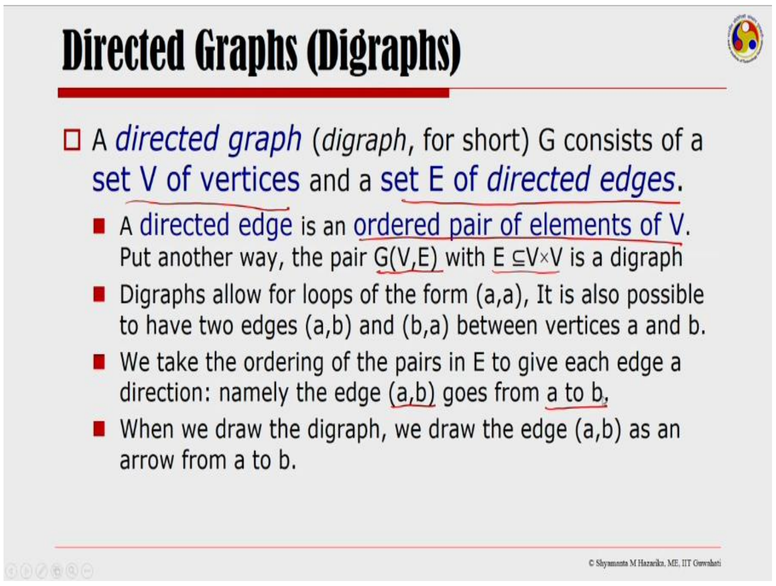
© Shyamanta M Hazarika, ME, IIT Guwahati

So, I am more interested here in finding out the path in a graph. Like if you recall we had this 8-puzzle tile configurations as where I explored and really got this sort of structure there, as I keep on expanding my nodes at every level. Given the 4 operators of the 8-puzzle game to me and from a given s as I could arrive at a given g. I am more interested in finding out how did I come from s to g or get to know what is a path.

So, the notion of a path in a graph is intuitively very clear. But, it is very difficult to pin down it formally. So, suppose I have a graph which is a set of vertices and edges and I have k vertices. So, 0, 1, 2, 3 up to k, k + 1; not necessarily this thing. And I have edges as e 1, e 2 so and so forth up to e to the power sub k; not necessarily this thing. Now, each edge AI is a group of vertices.

The alternating sequence that I get of the vertices and edges, starting from the vertice v sub 0 to v sub k is a path from v sub 0 to v sub k of length k. The length is the number of edges and it is not the number of vertices. So, for this path that I have highlighted here, the length of the path is 1, 2, 3 and 4. So, here is a 4 length path. And what I wanted to highlight was that this path is made up of edges e 1, e 2, e 3, e 4. So, alternatively it had vertices. Let us call this vertice as v a, v b, v c. So, the path from the start to the goal for the given graph would be something like s, e sub 1, v sub a, e sub 2, v sub b, e sub 3, v sub c, e sub 4 and g. So that is the path that is as in the given graph.

(Refer Slide Time: 10:43)



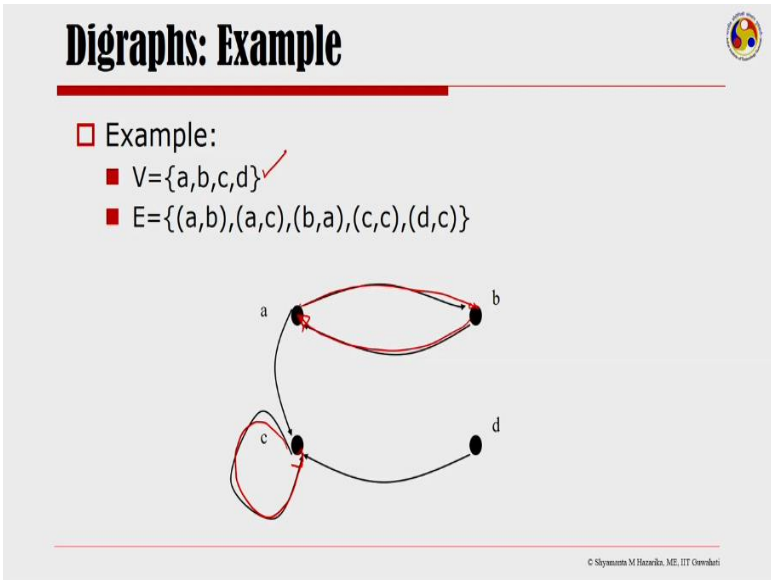
Directed Graphs (Digraphs)

- A *directed graph* (*digraph*, for short) G consists of a set V of vertices and a set E of *directed edges*.
  - A directed edge is an ordered pair of elements of V. Put another way, the pair G(V,E) with E ⊆V×V is a digraph
  - Digraphs allow for loops of the form (a,a), It is also possible to have two edges (a,b) and (b,a) between vertices a and b.
  - We take the ordering of the pairs in E to give each edge a direction: namely the edge (a,b) goes from a to b.
  - When we draw the digraph, we draw the edge (a,b) as an arrow from a to b.

© Shyamanta M Hazarika, ME, IIT Guwahati

What we are more interested when we were looking at graph search for solutions to given problems is what are called directed graphs or digraphs for short. Here, again it has a set of vertices as V. But the set of edges is actually directed edges. So, what are directed edges? Directed edges are ordered pair of elements of V. Put another way, if I have a graph of 2 tuple of V and E; E is a subset of V cross V. It is a digraph if these pair which I get are ordered pairs. So, ordering of pairs in E to give each as a direction, namely the edge a b goes from a to b.

(Refer Slide Time: 11:32)



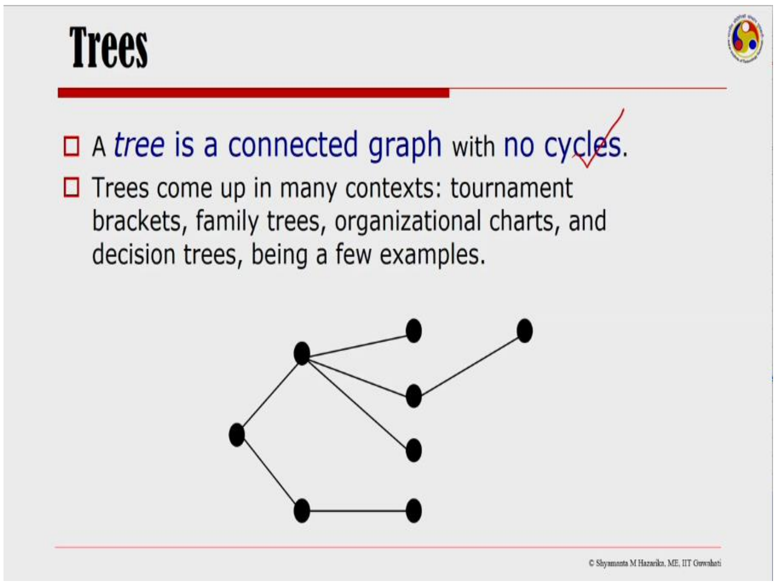
Digraphs: Example

- Example:
  - V={a,b,c,d}
  - E={(a,b),(a,c),(b,a),(c,c),(d,c)}

© Shyamanta M Hazarika, ME, IIT Guwahati

So, here is an example of a digraph. I have a set of vertices, a, b, c, d and pair of edges a b, a c, b a, c c, d c. So, if these pairs are directed and arcs rather than just being edges as in the general graph that I was talking of, then what we have is a digraph.

(Refer Slide Time: 11:57)



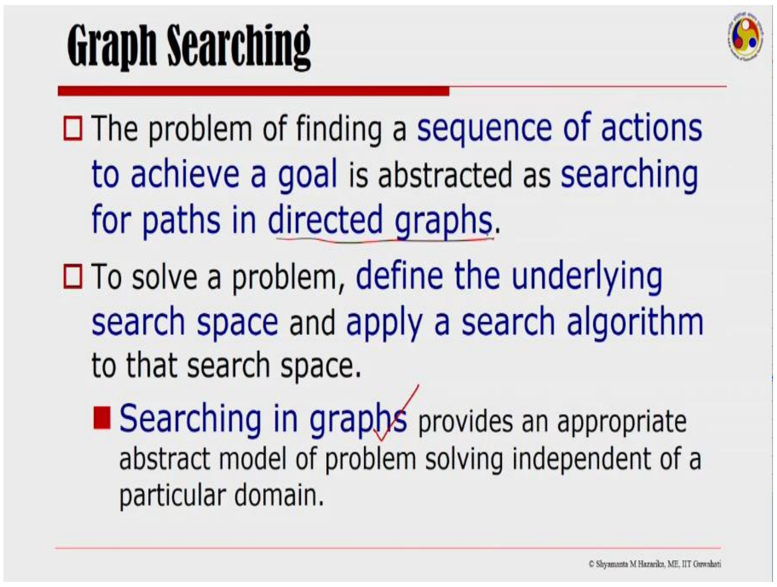
Trees

- A *tree* is a connected graph with no cycles.
- Trees come up in many contexts: tournament brackets, family trees, organizational charts, and decision trees, being a few examples.

© Shyamanta M Hazarika, ME, IIT Guwahati

A tree is a connected graph with no cycle. So here, if you see, if you go back and see the directed graph, here we have a arc that starts at c and comes back to c again. Or, if you could see here, we go from a to b and then there is a path from b to a. Such a thing is called a cycle. So, in a tree, we have a graph which is connected, but they do not have cycles. And this something that we will explore in the remainder of this lecture.

(Refer Slide Time: 12:33)



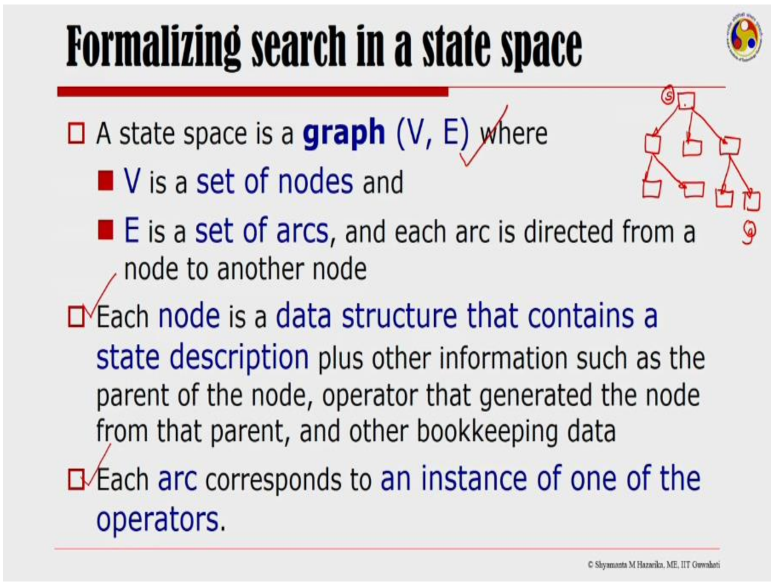
Graph Searching

- The problem of finding a sequence of actions to achieve a goal is abstracted as searching for paths in directed graphs.
- To solve a problem, define the underlying search space and apply a search algorithm to that search space.
  - Searching in graphs provides an appropriate abstract model of problem solving independent of a particular domain.

© Shyamanta M Hazarika, ME, IIT Guwahati

So, graph searching for problem solving as state space search is to find a sequence of actions to achieve a goal as searching for paths in a directed graph. To solve the problem, we define the underlying search space and then apply a search algorithm. Searching in graphs therefore provides an appropriate abstract model of problem solving, independent of the particular domain. And the type of graph we are searching here; one needs to remember is usually a directed graph.

(Refer Slide Time: 13:11)



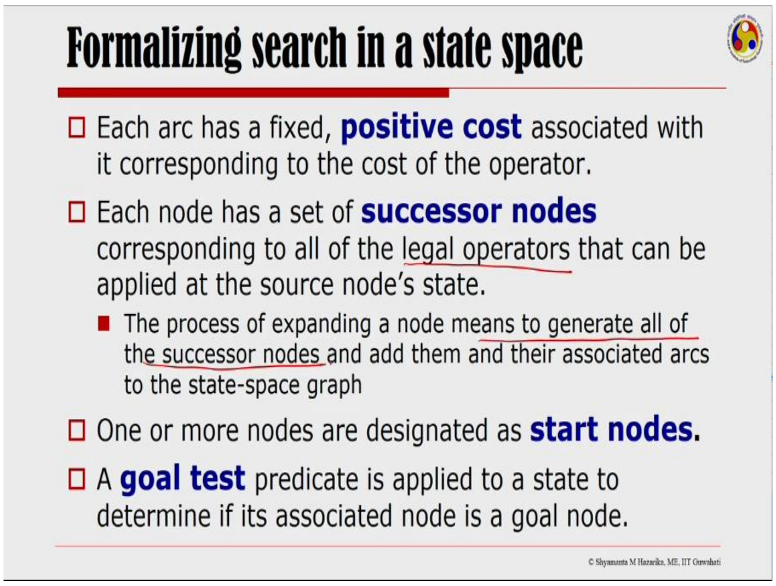
Formalizing search in a state space

- A state space is a **graph** (V, E) where
  - V is a set of nodes and
  - E is a set of arcs, and each arc is directed from a node to another node
- Each node is a data structure that contains a state description plus other information such as the parent of the node, operator that generated the node from that parent, and other bookkeeping data
- Each arc corresponds to an instance of one of the operators.

So, we are in now position to formalize search in state space. So, let us define a state space. A state space for us is actually a graph of a vertices and edges where V is a set of nodes and E is the set of arcs. Here when I say nodes, the node maybe a simple node or it could be a data structure that contains a state description. It could contain other information such as parent of the node, operator that generated that node from that parent and other bookkeeping data.

And each arch corresponds to an instance of one of the operators. So basically, whenever we are talking of formalizing search in a state space is we are looking at having whole of the state space converted into nodes with each operator giving me some instances that takes me to the other nodes. So, finally, we end up having such a directed graph. And searching in this directed graph for a path that brings me from s to g is what I mean by getting to the solution.

(Refer Slide Time: 14:33)



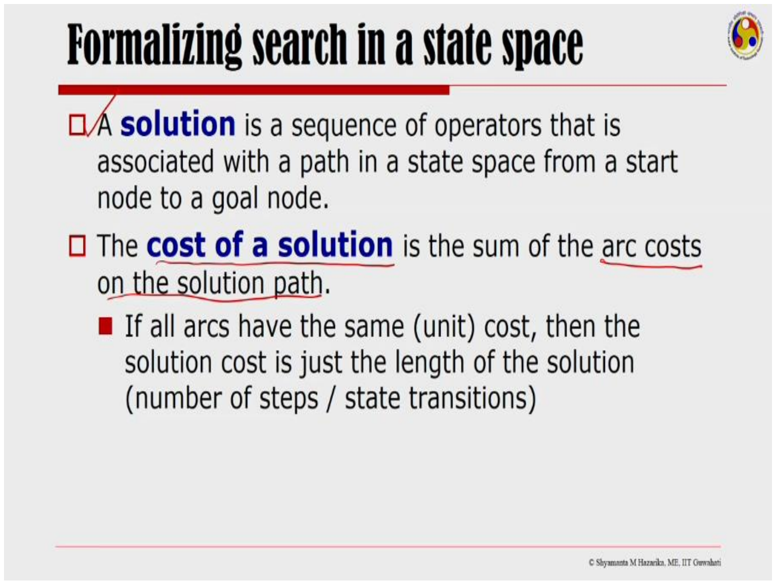
Formalizing search in a state space

- Each arc has a fixed, **positive cost** associated with it corresponding to the cost of the operator.
- Each node has a set of **successor nodes** corresponding to all of the legal operators that can be applied at the source node's state.
  - The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as **start nodes.**
- A **goal test** predicate is applied to a state to determine if its associated node is a goal node.

© Shyamanta M Hazarika, ME, IIT Guwahati

For each arc in such a directed graph has a fixed positive cost associated with it, corresponding to the cost of the operator. And each node has a set of successor nodes corresponding to all of the legal operators as that can be applied to the applied to the source node's state. The process of expanding a node means to generate all the successor nodes and add them to their associated arcs. 1 or more nodes are designated as the start node. And then there is a goal test predicate and which is applied to a state to determine if its associated node is a goal node.

(Refer Slide Time: 15:17)



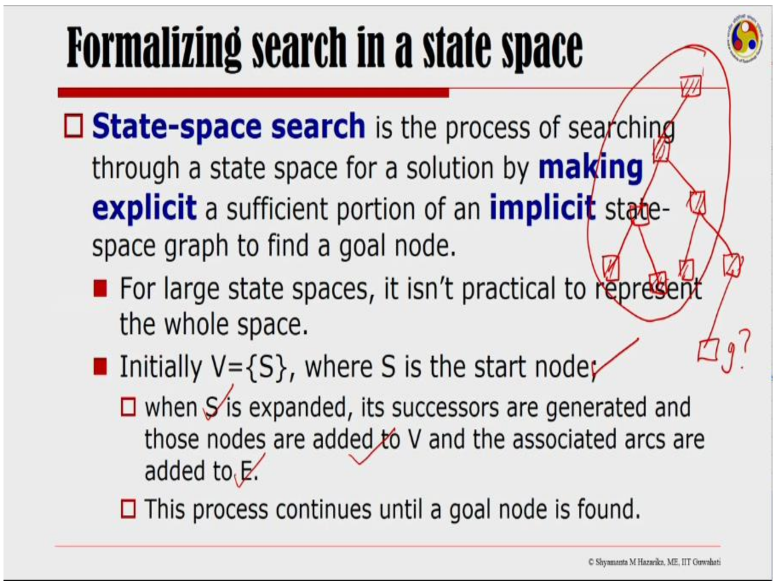
Formalizing search in a state space

- A **solution** is a sequence of operators that is associated with a path in a state space from a start node to a goal node.
- The **cost of a solution** is the sum of the arc costs on the solution path.
  - If all arcs have the same (unit) cost, then the solution cost is just the length of the solution (number of steps / state transitions)

© Shyamanta M Hazarika, ME, IIT Guwahati

A solution in such a formal framework is a sequence of operators that is associated with a path in a state space from a start node to a goal node. And the cost of each solution that I am looking for is the sum of the arc costs on the solution path. So, if all the arcs have the same unit cost, then the solution cost is just the length of the solution or the number of steps or the state transitions that is involved.

(Refer Slide Time: 15:51)



Formalizing search in a state space

- **State-space search** is the process of searching through a state space for a solution by **making explicit** a sufficient portion of an **implicit** state-space graph to find a goal node.
  - For large state spaces, it isn't practical to represent the whole space.
  - Initially V={S}, where S is the start node;
    - when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E.
    - This process continues until a goal node is found.

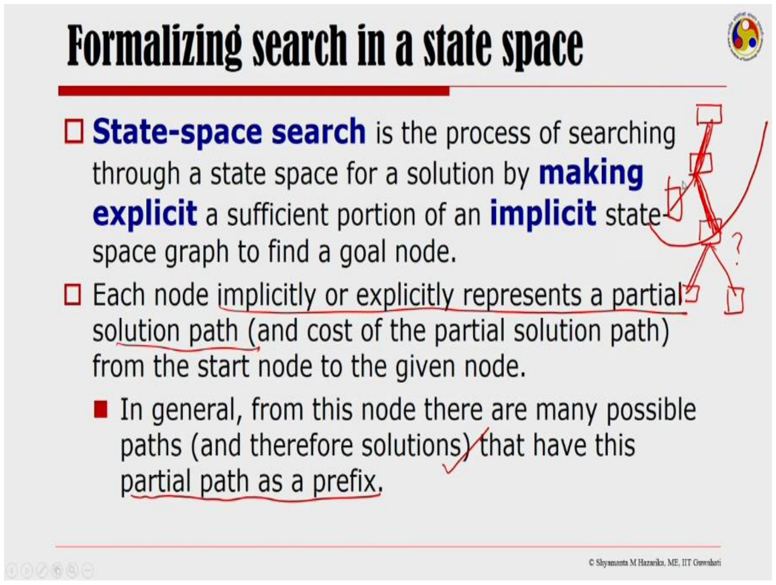
© Shyamanta M Hazarika, ME, IIT Guwahati

So, state space search under such a scenario is the process of searching through a state space for a solution, by making explicit a sufficient portion of an implicit state space graph. Now, one needs to remember that we do not generate the complete graph when we are looking for a solution under this idea of problem solving. We keep on generating the graph as we keep on exploring for the solution.

So, it is the process by which you search to a space, state space by making explicit only a sufficient portion of it; of an implicit state space graph to find a goal node. For large state spaces, it is important to realize that it is not practical to represent the whole space. So, you initially start with just the start node. And then expand the start node S. Its successors are generated and those nodes are added to the set V and the associated arcs are added to the set E.

This process continues until a goal node is found. So, instead of having the whole state space graph as it is explicitly generated and then looking for a solution, the idea is to make explicit a sufficient portion of this as I go on expanding for nodes. And every node that I expand answering a question whether that is my goal node; if not, getting its successor nodes and keeping the process containing.

(Refer Slide Time: 17:50)



Formalizing search in a state space

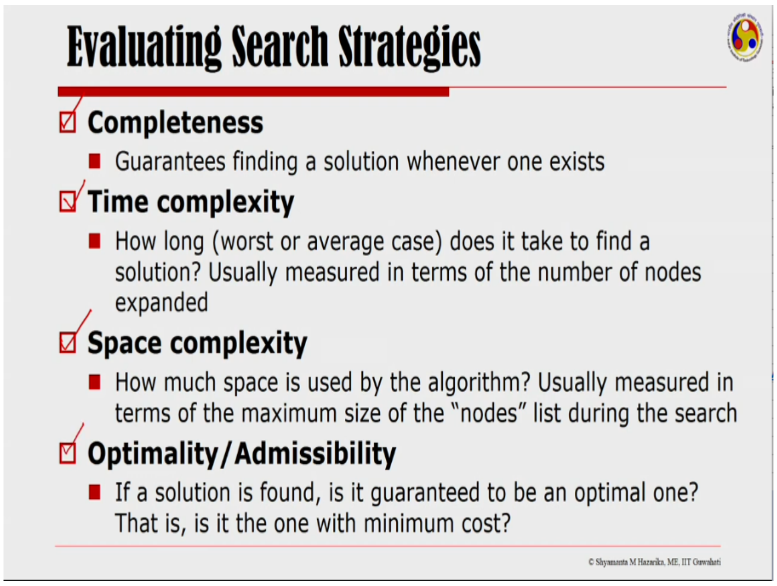
- **State-space search** is the process of searching through a state space for a solution by **making explicit** a sufficient portion of an **implicit** state-space graph to find a goal node.
- Each node implicitly or explicitly represents a partial solution path (and cost of the partial solution path) from the start node to the given node.
  - In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

© Shyamanta M Hazarika, ME, IIT Guwahati

So, state space search is when each node implicitly or explicitly represents a partial solution path. In general, from this node, there are many possible paths that have its partial path as a prefix. So, basically what it means is that when I am doing such a state space as exploration in a directed graph I have come up to this point. And then I keep on expanding beyond. But one needs to remember that whatever I have expanded up to this point, this still forms a part of the path.

So, it is still, the partial path is still there as a prefix for me. Beyond this, there would be many alternate paths, but this one up to here; if somewhere the goal lies below this node, then up to here, this is the partial path that is included already.

(Refer Slide Time: 18:49)



Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded
- **Space complexity**
  - How much space is used by the algorithm? Usually measured in terms of the maximum size of the "nodes" list during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

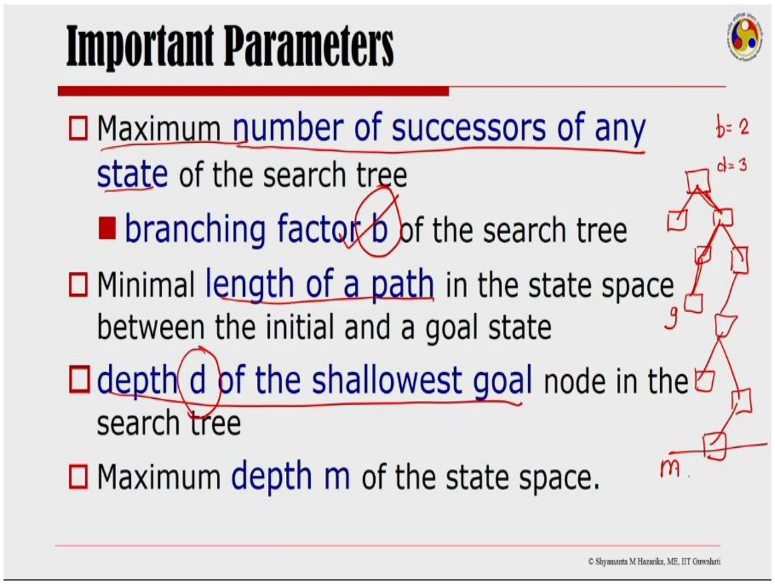
© Shyamanta M Hazarika, ME, IIT Guwahati

So, how do we evaluate such start search strategies? It is important to realize that we have many such searching techniques that we will discuss as which are uninformed. But, how do we evaluate such search strategies? There are 4 this thing matrixes. 1. We talk of completeness. Completeness is a guarantee of finding a solution whenever one exists. So, if a solution exists and if your searching technique can find that solution and the technique is said to be complete.

Next, we talk of time complexity. That is how long does it take to find a solution. This is usually measured in terms of the number of nodes that the searching technique expands. Next, we talk of space complexity. How much space is used by the algorithm. This is also usually measured in terms of the maximum size of the nodes list during the search. And then, there is a question of whether this search technique is optimal or admissible.

If a solution is found and is it guaranteed to be an optimal one? And that is, is it the one with the minimum cost? If that guarantees there for the given searching technique, then it is said to be optimal. We will now look at these searching techniques and then look at the evaluation of these techniques along these 4 matrices.

(Refer Slide Time: 20:33)



Important Parameters

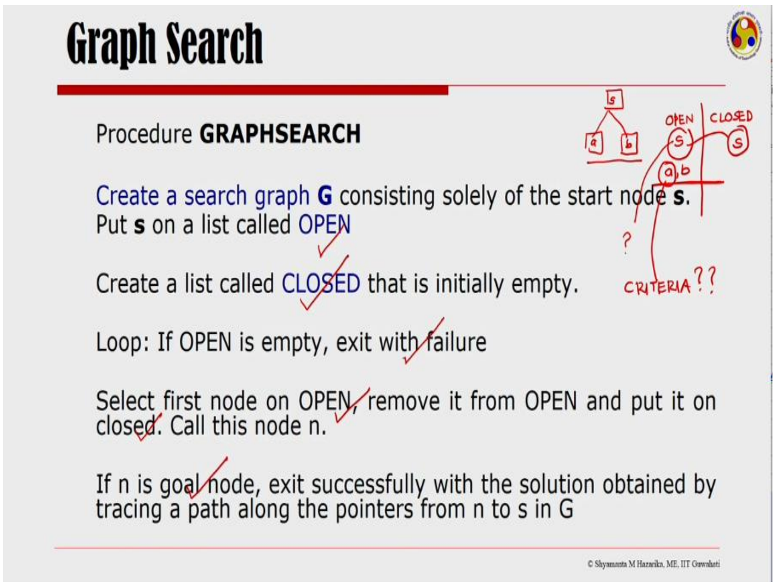
- Maximum number of successors of any state of the search tree
  - branching factor b of the search tree
- Minimal length of a path in the state space between the initial and a goal state
- depth d of the shallowest goal node in the search tree
- Maximum depth m of the state space.

b = 2
d = 3
g
m
© Shyamanta M Hazarika, ME, IIT Guwahati

So, what are the important parameters that one needs to really look at before we go into looking at the search techniques and evaluation of them is as these 3 interesting things that needs to be remembered. We talk of a number of successor of any state; the maximum number. That is called the branching factor. Then we talk of the length of a path in the state space as the minimal length which is the depth of the shallowest goal that is d.

So, we need to understand and something about d and something about b. So, if I have a node that is expanded every time into 2 successors; so, here my branching factor is 2. If my goal lies here, that is the depth of the minimum length of the path to the shallowest goal, then my d is 3. And I could have a maximum depth at which goes on it could be infinite. And then, maximum depth of the search state space is m.

(Refer Slide Time: 21:42)



s
a
b
OPEN
CLOSED
s
s
a, b
?
CRITERIA ??

Graph Search

Procedure **GRAPHSEARCH**

Create a search graph **G** consisting solely of the start node **s**. Put **s** on a list called OPEN

Create a list called CLOSED that is initially empty.

Loop: If OPEN is empty, exit with failure

Select first node on OPEN, remove it from OPEN and put it on closed. Call this node n.

If n is goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G

So, given this parameters to me and given the idea that we could search a directed graph to arrive at from a given and state to a given goal. We look at a graph search procedure. So, even before I write, there got in for the graph search procedure. I would like to highlight that what we literally do is we have the start node s and we have its successors. Let us call this successors a and b.

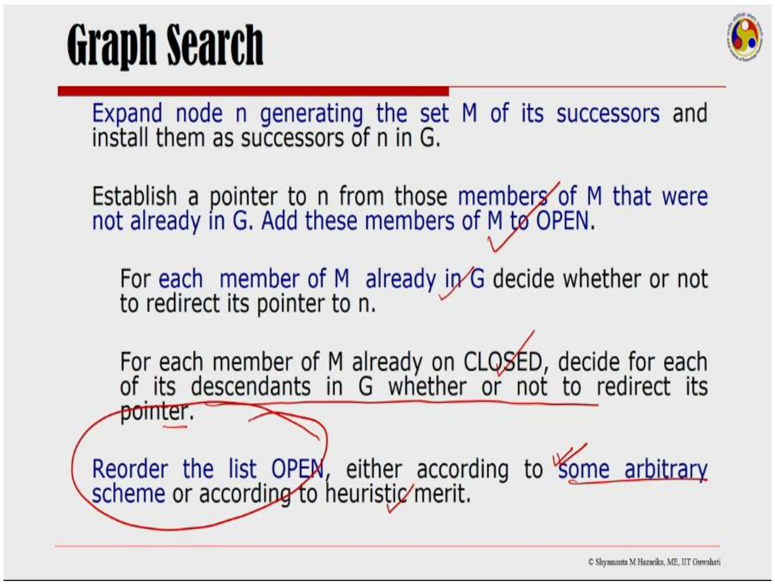
So, we create 2 lists. One called the open and other called the closed. And we first put s in the point where it needs to be expanded. So, we create a list called open and we put s in the open list. So, we put s here, consisting solely of the start node. And then we pick up s for expansion. So, we pickup s. And what we want to really check now is that, is s as the goal node itself? If not, we generates its successors.

So, if s is the goal node, we have nothing to do. We must come out and exit. That is what is a check here by saying that I create a list, a is called closed that is initially empty. And then, if there is nothing in open, I have to exit with failure. But, otherwise I select the first node on open and remove it on from open. I take it away from open and put that node in close. Now, why do I do that is a way to remember that s has been expanded.

So then, after that, what I do is I generate the successors of s. So, the successors of s here are a and b. So, I take a and b and put it in open. And next, when I look for an expansion, I take the node from open. So, a comes to me. Now, one needs to remember that how I add a b onto the open list will depend on what criteria do I really follow in order to really arrange this is. Later on, we will see that if I am doing a breadth-first search, I would see that everything is added to the end of the list.

If I am doing a depth-first search, I would add it to the beginning of the list. Okay. So basically, you create a list called closed that is initially empty and you take the first node and open. You remove it from open and put it on close. And then, if n is goal node, we have nothing to do, we exit immediately. Else, what we do is the following.

(Refer Slide Time: 24:45)



Graph Search

Expand node n generating the set M of its successors and install them as successors of n in G.

Establish a pointer to n from those members of M that were not already in G. Add these members of M to OPEN.

For each member of M already in G decide whether or not to redirect its pointer to n.

For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer.

Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

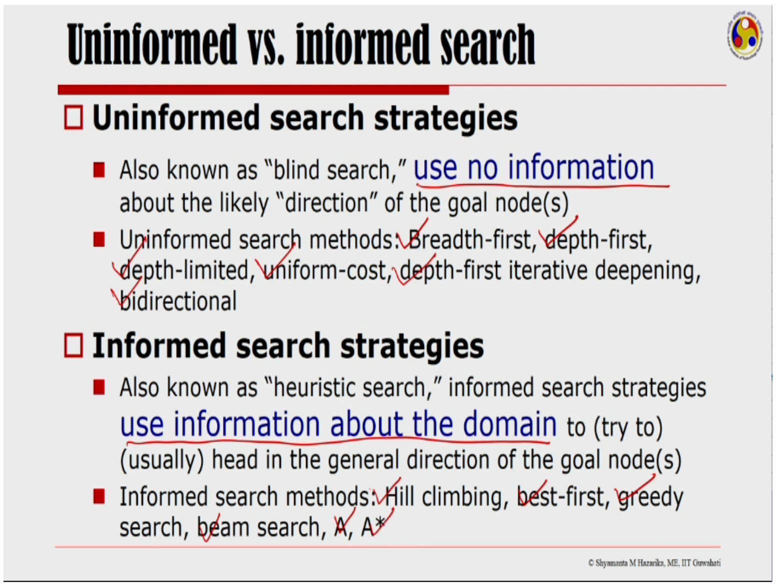
© Shyamanta M Hazarika, ME, IIT Guwahati

We expand the node n generating the set M of its successors. So, we install them as successors of M in the open list. So, add this members of M to the open list. And then, expand them further. Now, if each member of M is already in G, then we can decide whether we want to or redirect its pointers. Or, if each member is already in close, that means one of the successor is already been expanded.

Then we can redirect its descendants, whatever are there to that pointer. Else we will ignore that node to be expanded further. So, how do we reorder the list open? And is very, very important to understand and the type of technique that we will have in the graph search. So, they are either according to some arbitrary scheme if it is uninformed search. And according to some heuristic merit, if it is a informed search.

Today, our concentration would be only on looking at arbitrary schemes of arrangement and of the successor nodes in the open list. So, we look at uninformed search today.

(Refer Slide Time: 26:04)



Uninformed vs. informed search

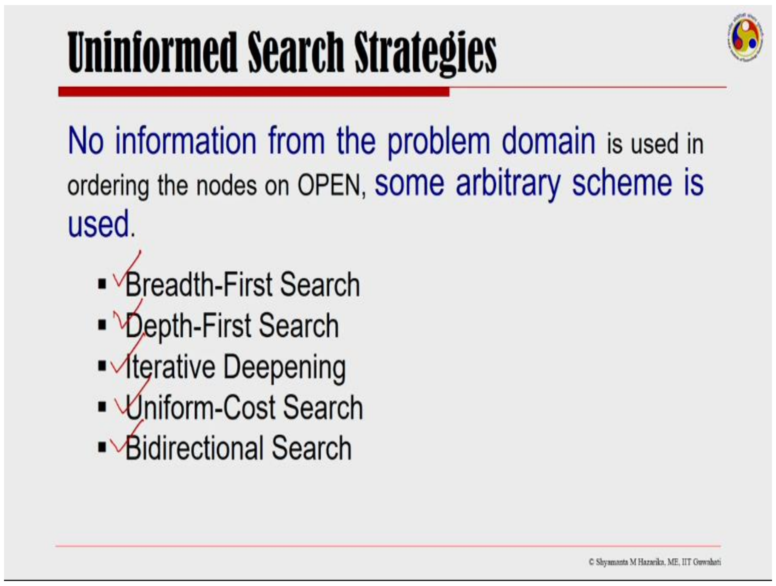
- Uninformed search strategies
  - Also known as "blind search," use no information about the likely "direction" of the goal node(s)
  - Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- Informed search strategies
  - Also known as "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)
  - Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*

© Shyamanta M Hazarika, ME, IIT Guwahati

And it is important that we really understand what we mean by uninformed here. Uninformed search is also called blind search because of the very fact that it does not use any information about the likely direction of the goal nodes. We do not have any idea of the problem domain. So, informed search on the other hand are also more popularly called heuristic search are informed search techniques in the sense that they use information about the domain and to try usually head in the general direction of the goal.

Informed search methods and uninformed search methods are distinctly different in only either use of no information or use of information about the domain. Few of the uninformed search methods are breadth-first search, depth-first search, depth-limited search, uniform-cost, depth-first iterative deepening and bidirectional search. We will look at few of them today. In informed search we have hill climbing, best-first, greedy search, beam search, A and A* which we will look at in the next lecture.

(Refer Slide Time: 27:26)



Uninformed Search Strategies

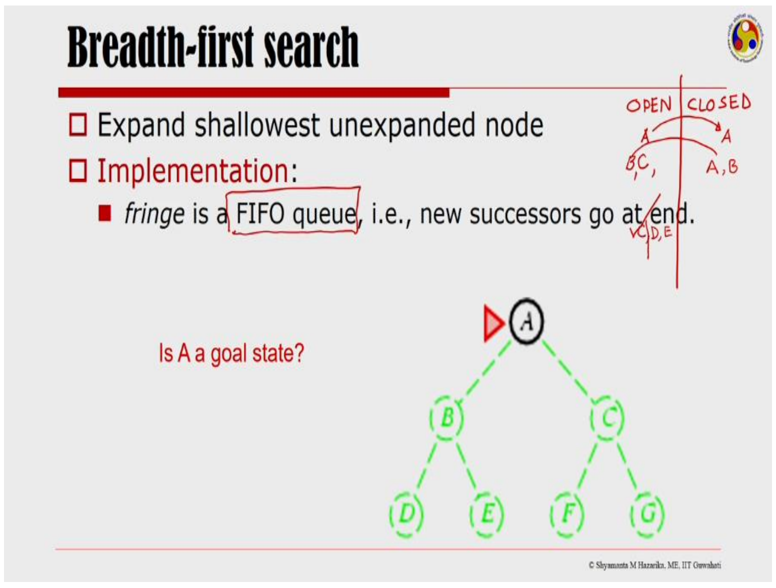
No information from the problem domain is used in ordering the nodes on OPEN, some arbitrary scheme is used.

- ✓Breadth-First Search
- ✓Depth-First Search
- ✓Iterative Deepening
- ✓Uniform-Cost Search
- ✓Bidirectional Search

© Shyamanta M Hazarika, ME, IIT Guwahati

So, uninformed search strategies; we do not have information from the problem domain. And that is how we order the nodes in open using some arbitrary scheme. Today we will look at this 5 uninformed search techniques: breadth-first, depth-first, iterative deepening, uniform-cost search and bidirectional search. So, what is breadth-first search. Let us look at it in more closer terms by using a real graph here.

(Refer Slide Time: 27:57)



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end.

Is A a goal state?

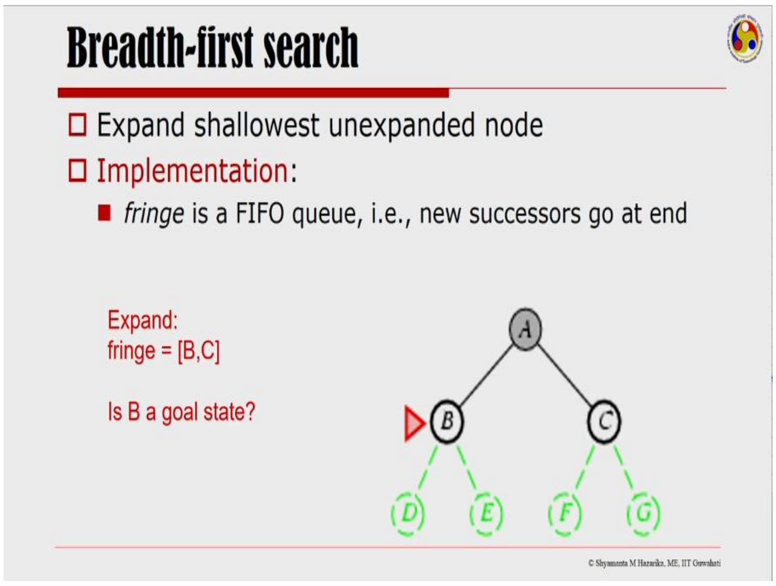
© Shyamanta M Hazarika, ME, IIT Guwahati

So, I have a graph with node a, b, c, d, e, f, g. So, what I expand is the shallowest unexpanded node. So, the fringe, if you can see by now is actually a first-in-first-out queue. So, new successors all go to the end. So, if you recall, we said we will create 2 lists when we are doing graph search; 1, an open list and another a closed list and we will keep the start node in open. As we expand, we take A to the closed list and generate the successors of A.

So, the successors of A that I get, I have B and C. Then, the next time I take B for expansion. Take B here and keep it in close. And when I get the successors of B; so, the successors of B equal C are D and E. So, those successors will go to the end of the list. So, D and E will come here. So, that the next node that I take for expansion is C rather than end D. So, this is how the breadth-first search works.

I expand the slowest unexpanded node. The fringe is a first-in-first-out queue. That is, the new successors go at the end. So, first I check if A is a goal state.

(Refer Slide Time: 29:23)



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

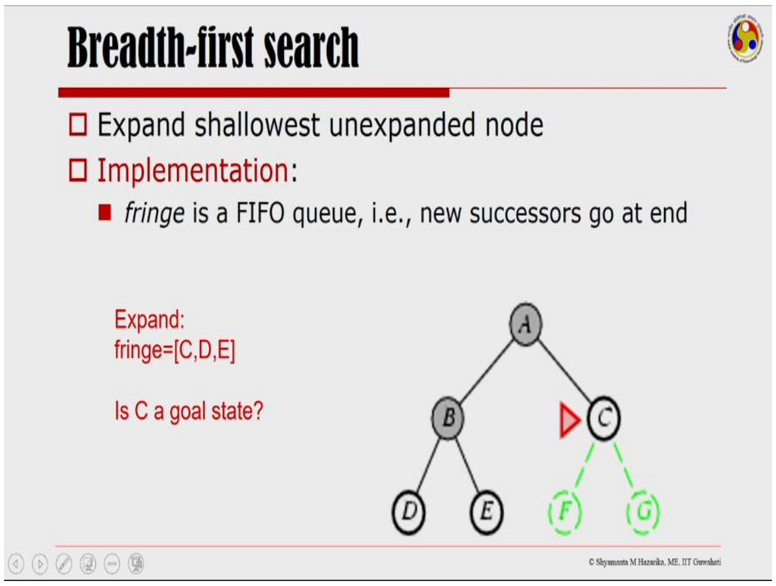
Expand:
fringe = [B,C]

Is B a goal state?

© Shyamanta M Hazarika, ME, IIT Guwahati

If A is not a goal state, I get its successors which is B and C. So, the fringe for me is B C now. So, next I check if B is a goal node.

(Refer Slide Time: 29:34)



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

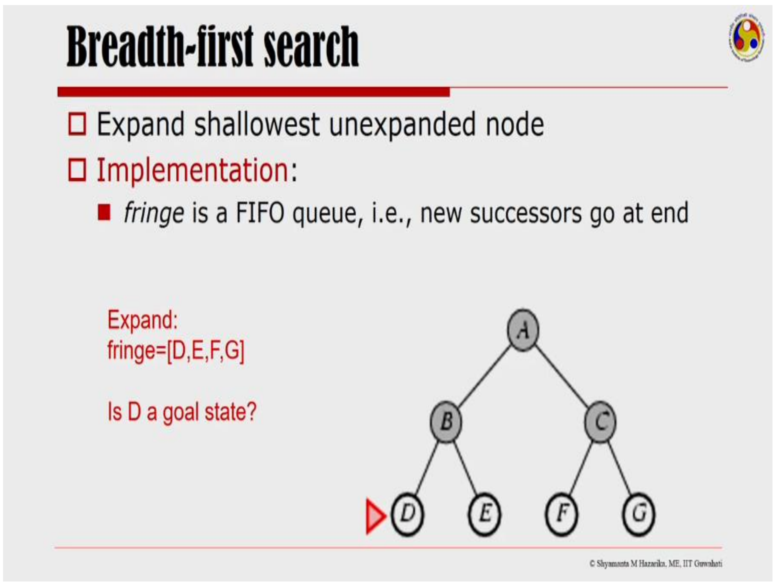
Expand:
fringe=[C,D,E]

Is C a goal state?

A
B
C
D
E
F
G
© Shyamanta M Hazarika, ME, IIT Guwahati

If not, I get its expansion which is D E. But then, the next node to be expanded is C and not D. That that is something that one needs to really understand here, when I am doing a breadth-first search.

(Refer Slide Time: 29:45)



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

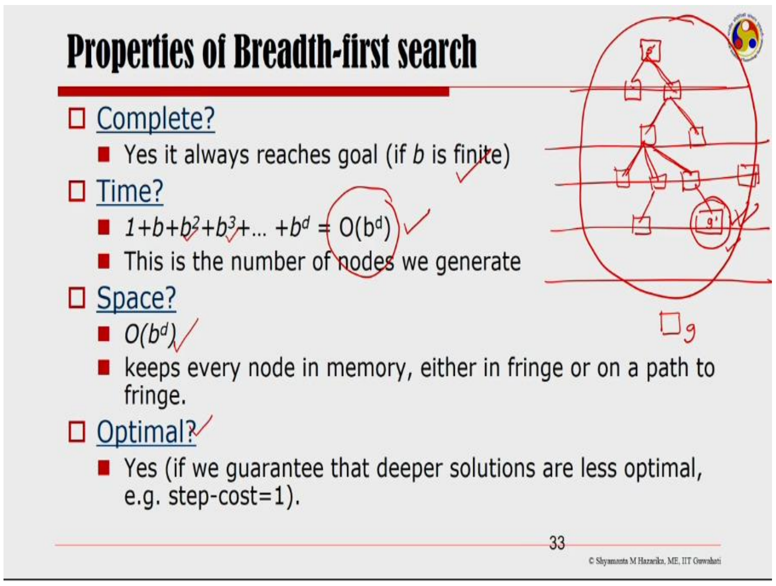
Expand:
fringe=[D,E,F,G]

Is D a goal state?

A
B
C
D
E
F
G
© Shyamanta M Hazarika, ME, IIT Guwahati

And then, next node to be expanded becomes D; and so on and so forth. So, a few questions that I want to answer about the properties of the breadth-first search.

(Refer Slide Time: 29:53)



Properties of Breadth-first search

- Complete?
  - Yes it always reaches goal (if *b* is finite)
- Time?
  - $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
  - This is the number of nodes we generate
- Space?
  - $O(b^d)$
  - keeps every node in memory, either in fringe or on a path to fringe.
- Optimal?
  - Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

33

© Shyamanta M Hazarika, ME, IIT Guwahati

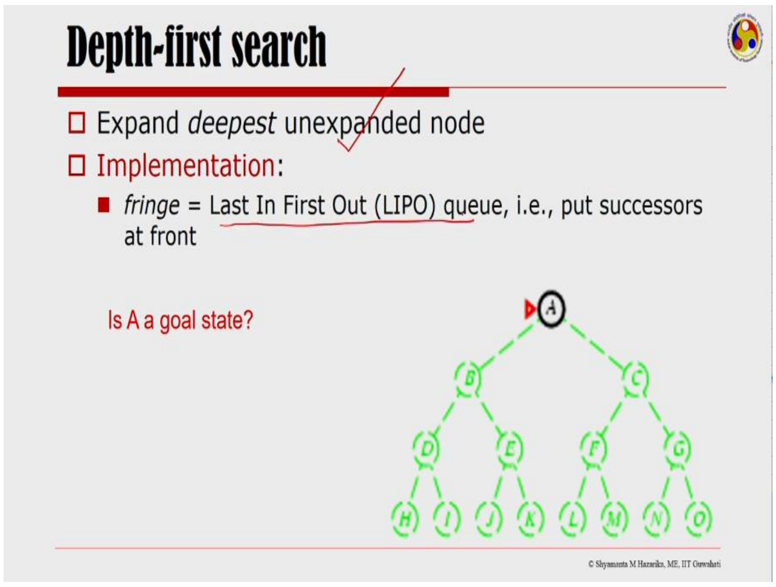
The first question is: Is breadth-first search complete? We can see that the breadth-first search always guarantees that it will reach the goal if the branching is finite. So, if I have a state space search to be done starting at a given s to some given g and I know that this is the state space within which I have all the configurations for those states. And then, if branching is finite for this search; so, at some point, somewhere, I will definitely arrive at g.

Why would that happen intuitively is because I would be expanding everything at every level. And that will guarantee that no state is missed for a finite B. And therefore, breadth-first search is complete. The number of nodes that we will generate is at this level, it is 1. And if I have b branching at this it is b, at this level it will become b square, then b cube. So, I would have an order of b to the power d.

This is the number of nodes we will generate. What about space complexity? Because of the very fact that the breadth-first search generates every node at every level, it has to keep all the nodes in it, that it had expanded. In case it keeps every node in the memory, either in fringe or on a path to the fringe. And therefore, the order is of b to the power d. Whether breadth-first search is optimal; now this depends if we guarantee the deeper solutions are less optimal, that is if step-cost is 1.

If we assume that every step it just costs 1 and there is no other cost involved. So, then, it would mean that, if I have some solution which is deeper down there and if I have a solution which is somewhere at a shallower node, I will get the shallower one first. And therefore, every time I am guaranteed to get the minimum cost solution. So, it is optimum.

(Refer Slide Time: 32:15)



Depth-first search

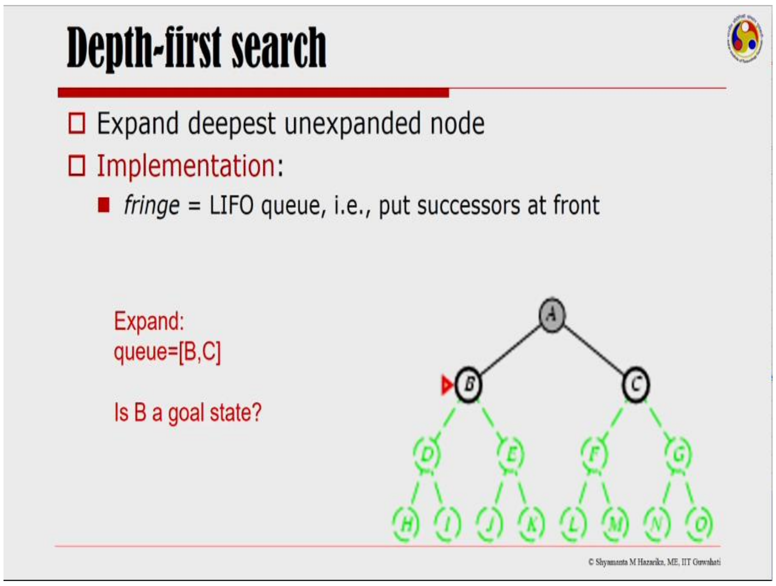
- Expand *deepest* unexpanded node
- Implementation:
  - *fringe* = Last In First Out (LIPO) queue, i.e., put successors at front

Is A a goal state?

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
© Shyamanta M Hazarika, ME, IIT Guwahati

Next, we will look at the depth-first search. So, you expand the deepest unexpanded node. That is what the clue is for depth-first search. And if you have realized by this time, I am, the fringe that I am talking of is last-in-first-out queue. That is, you put successors at the front and those are what is expanded. So, you first check if A is a goal node.

(Refer Slide Time: 32:40)



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

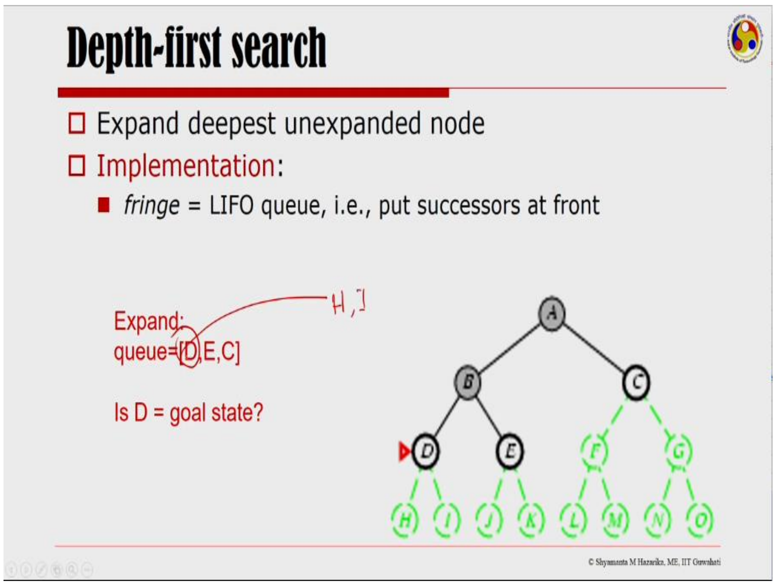
Expand:
queue=[B,C]

Is B a goal state?

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
© Shyamanta M Hazarika, ME, IIT Guwahati

Or, if not, you get the successors which are B and C and check if B is goal. If not, you expand that and you have D E. But the point here that I want to make is D E would be put in the front of the open list rather than at the end of the list in open.

(Refer Slide Time: 32:58)



Depth-first search

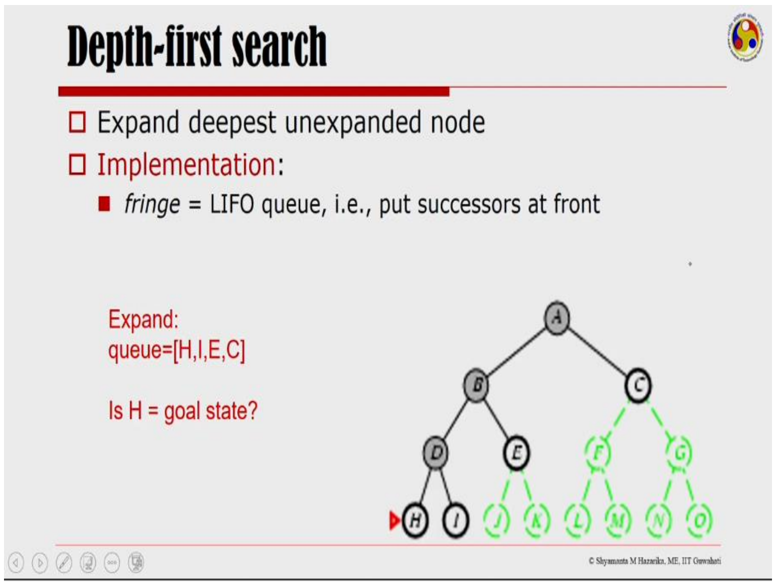
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[D,E,C]

Is D = goal state?

So, I have D. I checked if D is a goal state. And then, I get its successor H I.

(Refer Slide Time: 33:08)



Depth-first search

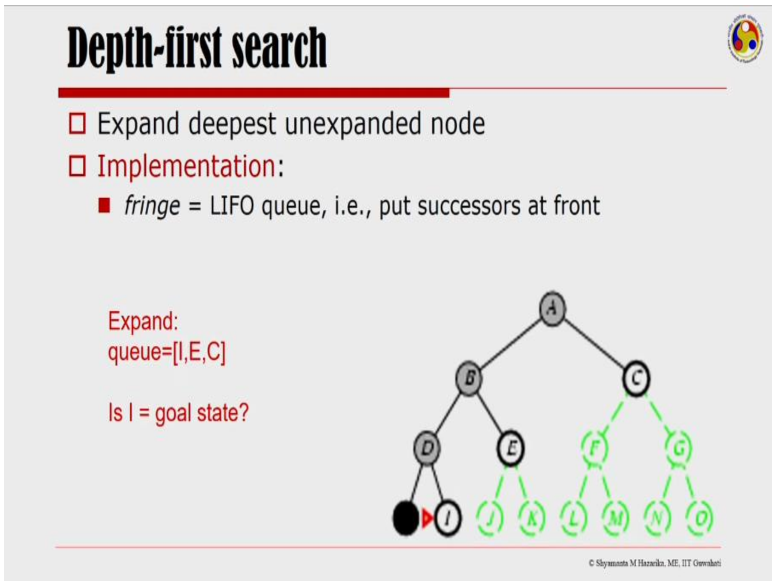
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[H,I,E,C]

Is H = goal state?

So, I put its successors H I in the front. So, here is the queue. If you go back 1 slide there, I had D which I expanded and D gave me 2 successors which was H and I. H and I would come to the front of the list rather than to the back of the list. So, I check and continue the search.

(Refer Slide Time: 33:25)



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

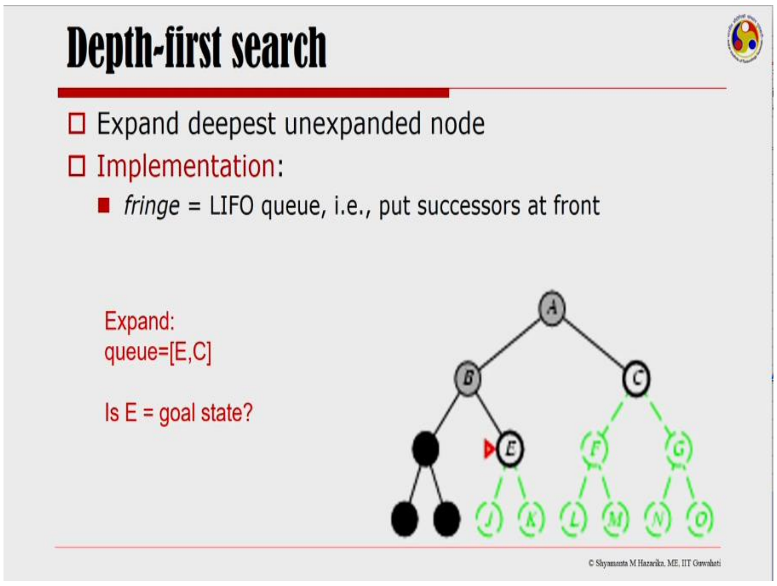
Expand:
queue=[I,E,C]

Is I = goal state?

© Shyamanta M Hazarika, ME, IIT Guwahati

Go on doing I;

(Refer Slide Time: 33:27)



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[E,C]

Is E = goal state?

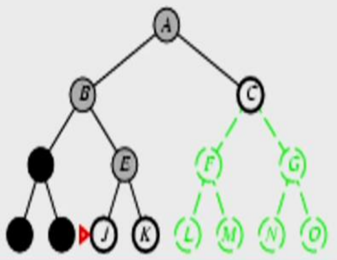
© Shyamanta M Hazarika, ME, IIT Guwahati

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[J,K,C]

Is J = goal state?

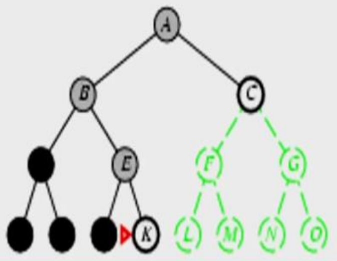


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[K,C]

Is K = goal state?

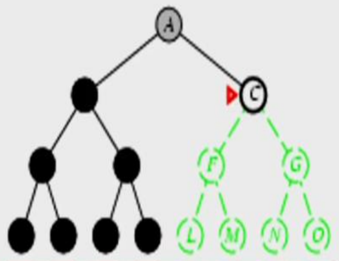


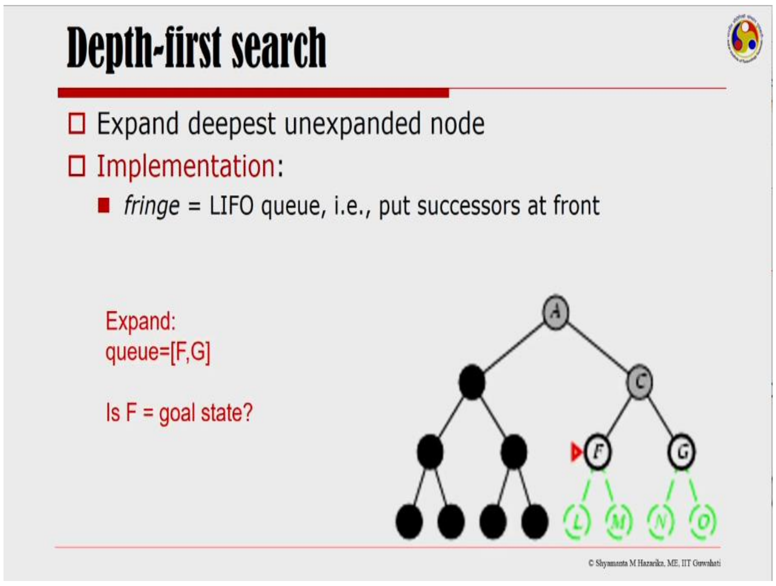
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[C]

Is C = goal state?





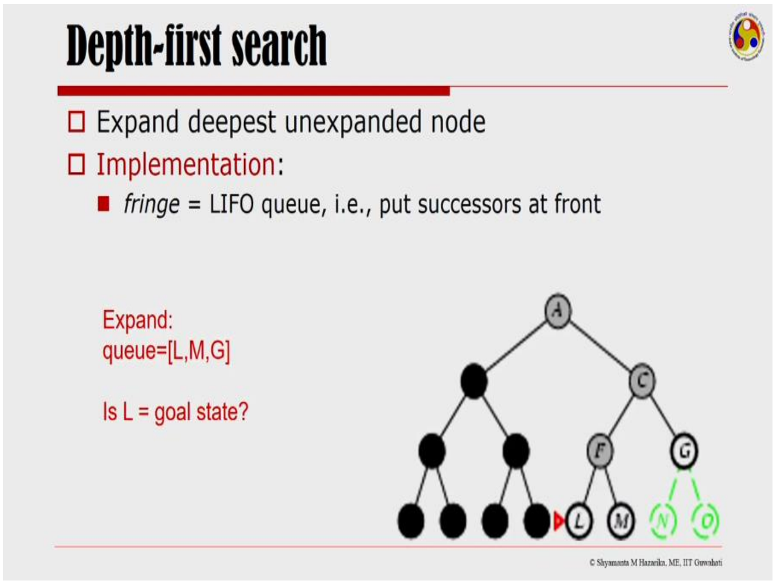
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[F,G]

Is F = goal state?

© Shyamanta M Hazarika, ME, IIT Guwahati



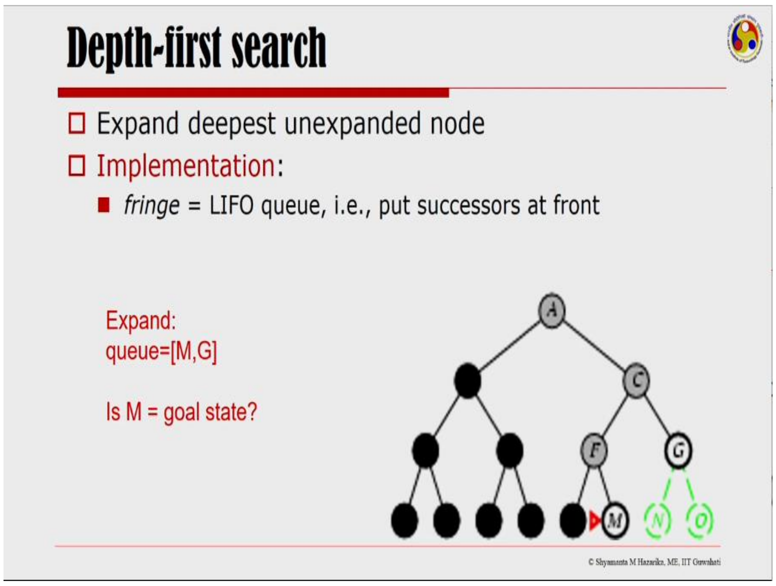
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

Expand:
queue=[L,M,G]

Is L = goal state?

© Shyamanta M Hazarika, ME, IIT Guwahati



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

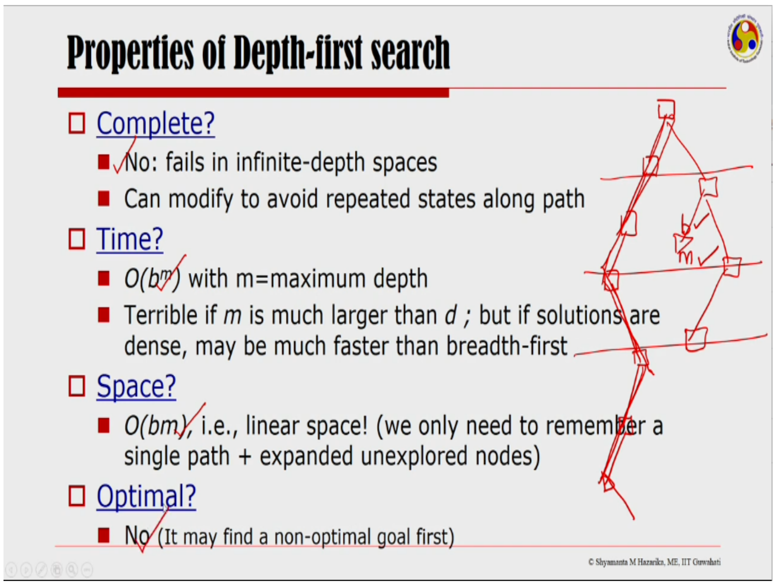
Expand:
queue=[M,G]

Is M = goal state?

© Shyamanta M Hazarika, ME, IIT Guwahati

And then go on doing everything till I expand at every level.

(Refer Slide Time: 33:32)



Properties of Depth-first search

- Complete?
  - No: fails in infinite-depth spaces
  - Can modify to avoid repeated states along path
- Time?
  - $O(b^m)$ with m=maximum depth
  - Terrible if m is much larger than d ; but if solutions are dense, may be much faster than breadth-first
- Space?
  - $O(bm)$, i.e., linear space! (we only need to remember a single path + expanded unexplored nodes)
- Optimal?
  - No (It may find a non-optimal goal first)

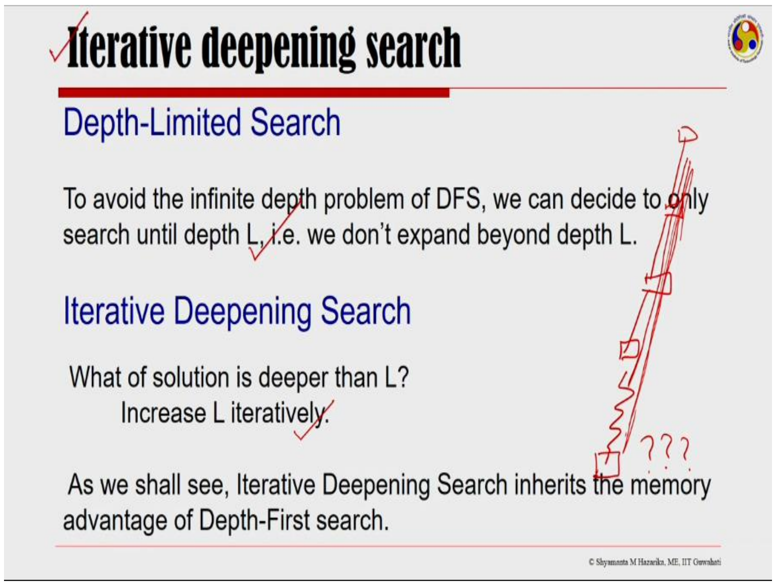
© Shyamanta M Hazarika, ME, IIT Guwahati

So, the questions of properties of depth-first search: Is depth-first search complete? You could see from these explanations that at depth-first search is not complete. Because, what can happen is that, if I have infinite depth space and I could be stuck in an allay, I could go on expanding down there a line and nowhere on this line a solution would exist. So, first important realization depth-first search is not complete.

What about the time complexity of depth-first search? If I have m which is the maximum depth and b being the branching factor, then the number of nodes that I need to do is b to the power m. And terrible if m is much larger than b. But if solutions and dense, may be much faster than breadth-first search. But, if I have a huge m, then it would be terrible time complexity. In terms of space, is definitely, we only need to remember a single path of unexpected and unexplored nodes.

So, the time complexity is b m which is linear. And it may find a non-optimal goal first. But, in terms of optimality, it could be possible intuitively that there are goals here somewhere at lower levels, which will definitely be missed by any depth-first search technique. So, it is not an optimal search technique.

(Refer Slide Time: 35:08)



Iterative deepening search

Depth-Limited Search

To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don't expand beyond depth L.

Iterative Deepening Search

What of solution is deeper than L?
Increase L iteratively.

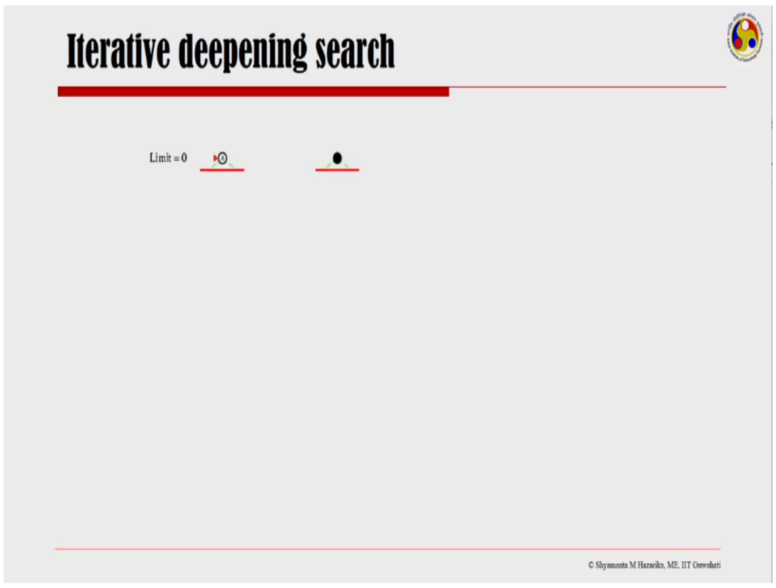
As we shall see, Iterative Deepening Search inherits the memory advantage of Depth-First search.

© Shyamanta M Hazarika, ME, IIT Guwahati

Next, we come to a modification of depth-first search which is depth-limited search. To avoid the infinite depth problem that I was showing you in the previous slide of depth-first search, we can decide to only search until a depth L; that is, we do not expand beyond depth L. But then, what will happen is that, if we keep that limit L, we may have a goal which lies just little well beyond L and we will never be able to find this.

So, a better idea would be that, what if I keep on changing L and interactively iteratively keep on increasing L. So, this is the idea of the next search technique that we will discuss called the iterative deepening search.

(Refer Slide Time: 36:09)

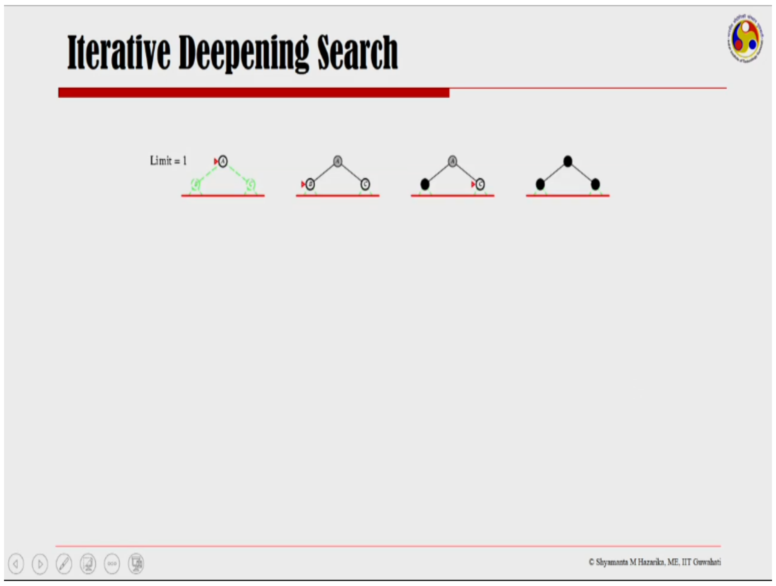


Iterative deepening search

© Shyamanta M Hazarika, ME, IIT Guwahati

So, iterative deepening search, I looked at level 0 first.

(Refer Slide Time: 36:11)

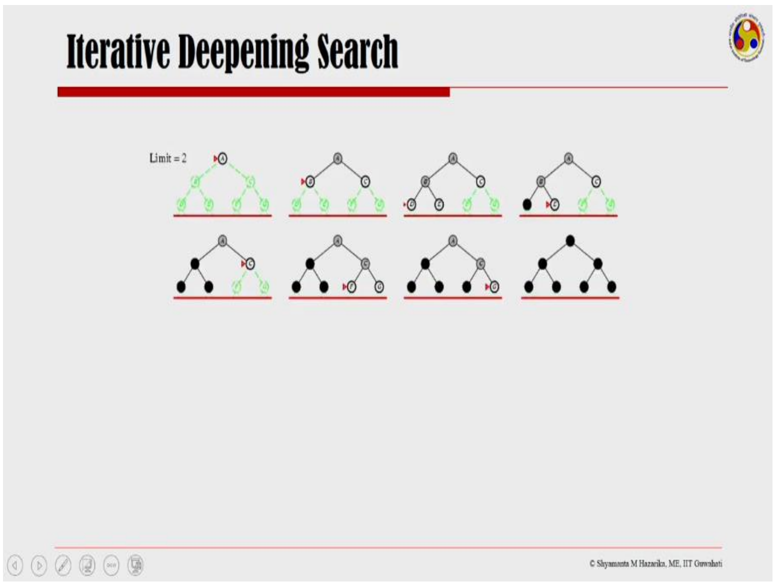


Iterative Deepening Search

© Shyamanta M Hazarika, ME, IIT Guwahati

Then I look at level 1 for the goal.

(Refer Slide Time: 36:13)

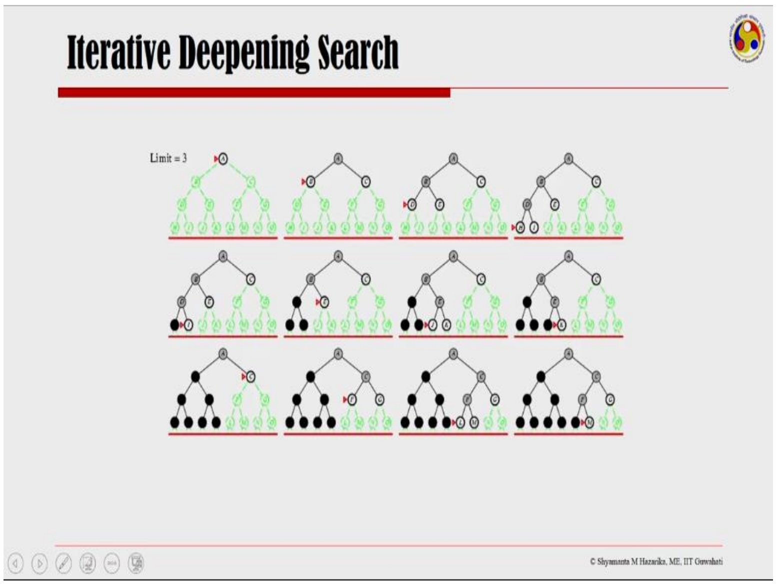


Iterative Deepening Search

© Shyamanta M Hazarika, ME, IIT Guwahati

Then level 2.

(Refer Slide Time: 36:15)

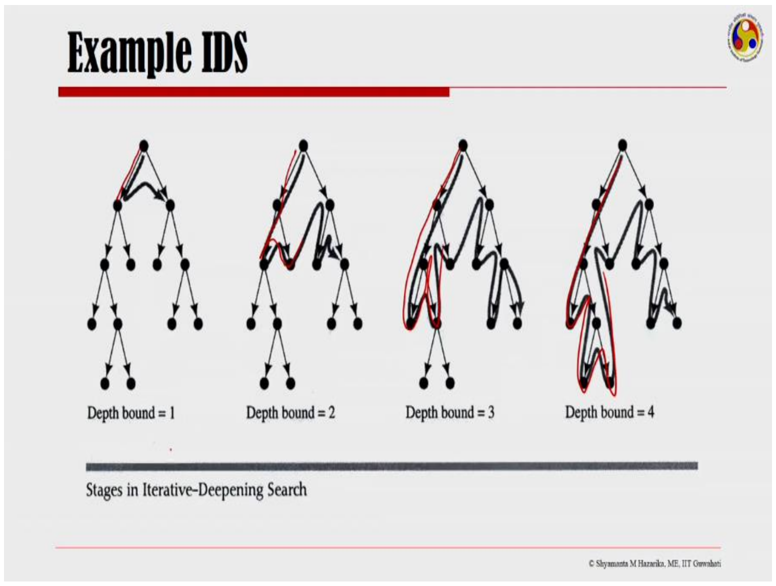


Iterative Deepening Search

© Shyamanta M Hazarika, ME, IIT Guwahati

Level 3, so on and so forth.

(Refer Slide Time: 36:17)



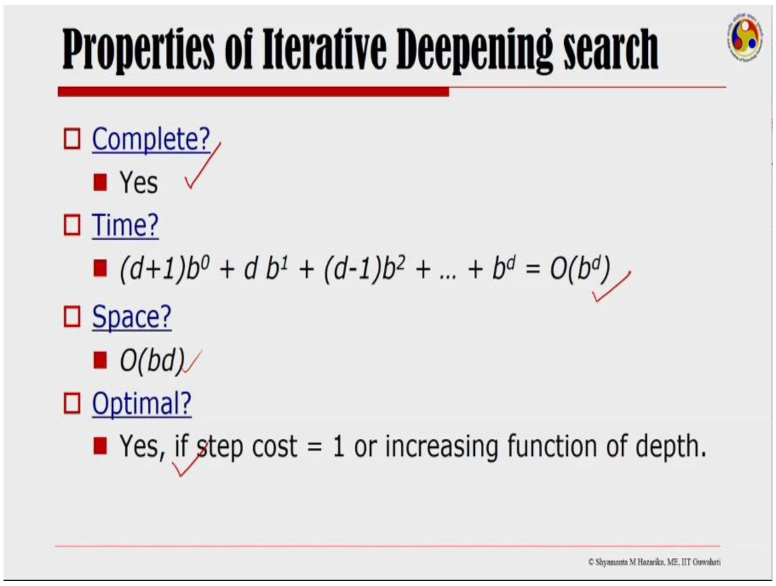
Example IDS

Stages in Iterative-Deepening Search

© Shyamanta M Hazarika, ME, IIT Guwahati

So, the idea of iterative deepening is, let me repeat it. So, I start with it level 1, then I go to level 2, level 3, level 4. So, the idea is that I take this level. I did not find things here. So, I thought of doing a little bit more deeper level. I did not find things there. I may think of going even down. I did not find things there. I may think of going even down. So, this is what is iterative deepening search, where I increased the depth bound at every iteration.

(Refer Slide Time: 36:53)



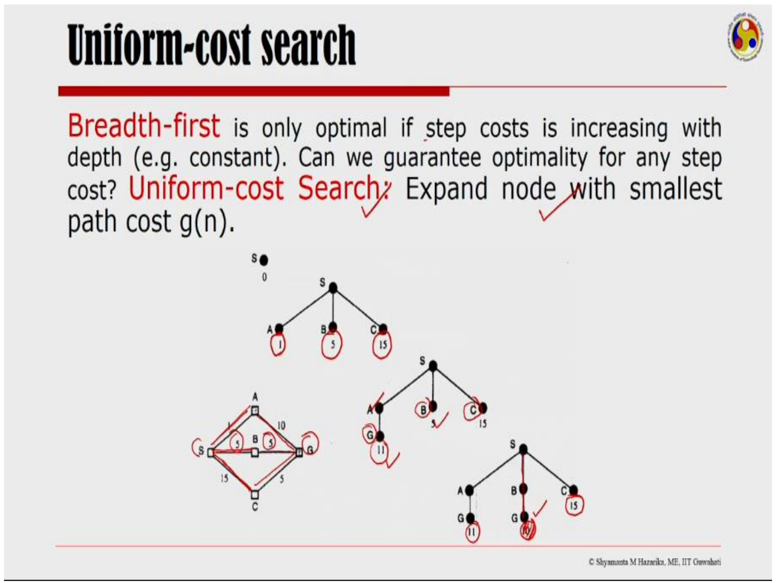
Properties of Iterative Deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost = 1 or increasing function of depth.

© Shyamanta M Hazarika, ME, IIT Guwahati

So, properties of iterative deepening, very quickly it is complete. The order of a time complexity is O, b to the power d. Space is O bd. And it is optimal if step cost is 1 or increasing function of depth.

(Refer Slide Time: 37:10)



Uniform-cost search

Breadth-first is only optimal if step costs is increasing with depth (e.g. constant). Can we guarantee optimality for any step cost? Uniform-cost Search: Expand node with smallest path cost g(n).

© Shyamanta M Hazarika, ME, IIT Guwahati

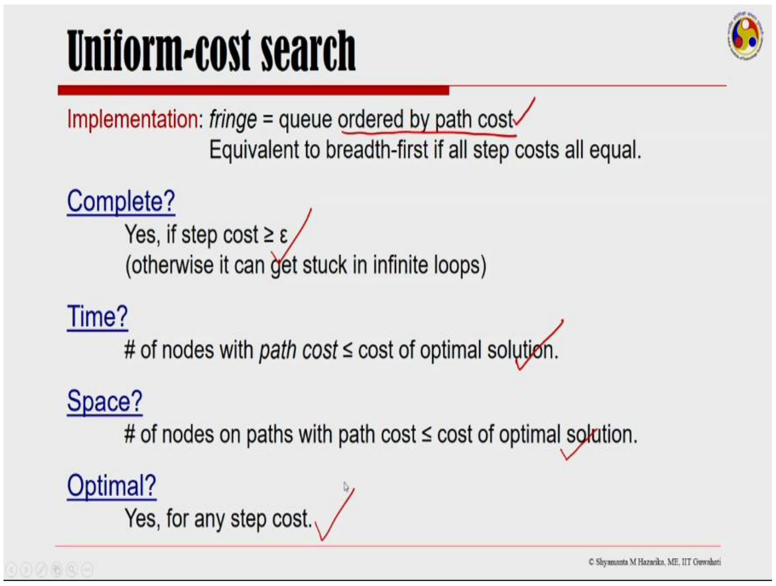
Next, we concentrate on something called a uniform-cost search. If you remember breadth-first is optimal. But this is optimal if step-cost is in; if only optimal if step-cost is increasing with depth. That is, it remains constant. Can we guarantee optimally for any step cost? For that, we look at something called the uniform-cost search. We expand node with the smallest path cost. So, here is a small graph.

This is a graph to find a path from s to g. So, here is my start and here is my goal. So, first when I expand s, I have then its successors A, B and C. But the question is, which one do I expand next. As when I look at this, I could see that the cost to A was 1, the cost to B was 5 and the cost to C was 15. So, I would rather expand the node with the smallest cost. So, I expand A.

So, my next expansion is of A where I get to G. And I know that the cost is A to G is 10 + 1 here. So, 10 + 1, 11. That is what I mark at G. And the other nodes that I have is B with a cost of 5 already there. And C with a cost of 15 already from the previous expansion. So, next which node to expand? If I ask the same question, then I go to expand the nodes in the smallest path. And therefore, I go and expand B.

So, once I expand B, I have here on this mark B to G. So, I get G. But the path cost is 5 and 5. That makes 10. So, the G cost is 10. And here it was 11. And now C 15. So, the next node that I want to expand is this node. And that, when I see that, I know that this is the goal node and I stop; but I pick up this. So, this path, that I got is the path which is with the minimum cost.

(Refer Slide Time: 39:29)



Uniform-cost search

Implementation: *fringe* = queue ordered by path cost
Equivalent to breadth-first if all step costs all equal.

Complete?
Yes, if step cost ≥ ε
(otherwise it can get stuck in infinite loops)

Time?
# of nodes with *path cost* ≤ cost of optimal solution.

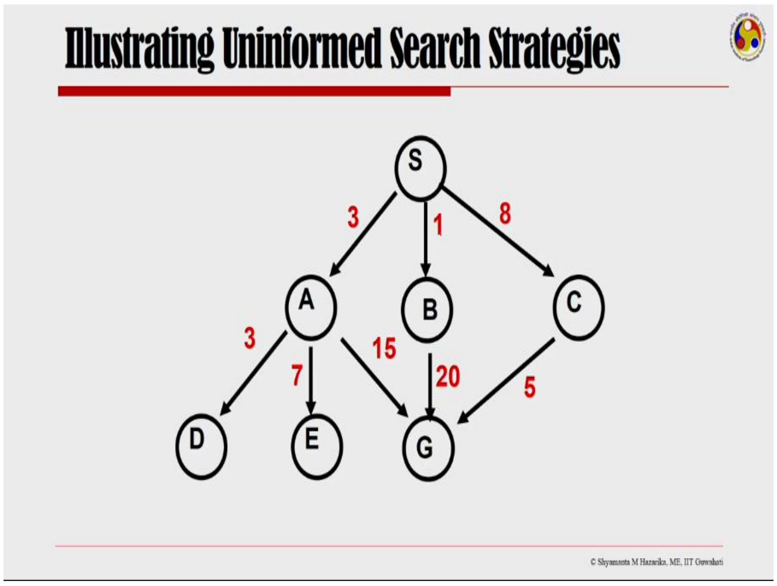
Space?
# of nodes on paths with path cost ≤ cost of optimal solution.

Optimal?
Yes, for any step cost.

© Shyamanta M Hazarika, ME, IIT Guwahati

So, uniform-cost search, implementation wise, the fringe that I am looking for is definitely queue. But then, they are ordered by the path cost. So, this is equivalent to breadth-first if all step costs were all equal. The question is whether they are complete. Yes, they are. And the number of nodes with path cost is always less than equal to cost of optimal cost number of nodes on path with path cost is greater than equal to cost of optimal cost. And whether it is optimal or not. Yes, for any step cost the uniform-cost search is an optimal graph search technique.

(Refer Slide Time: 40:09)

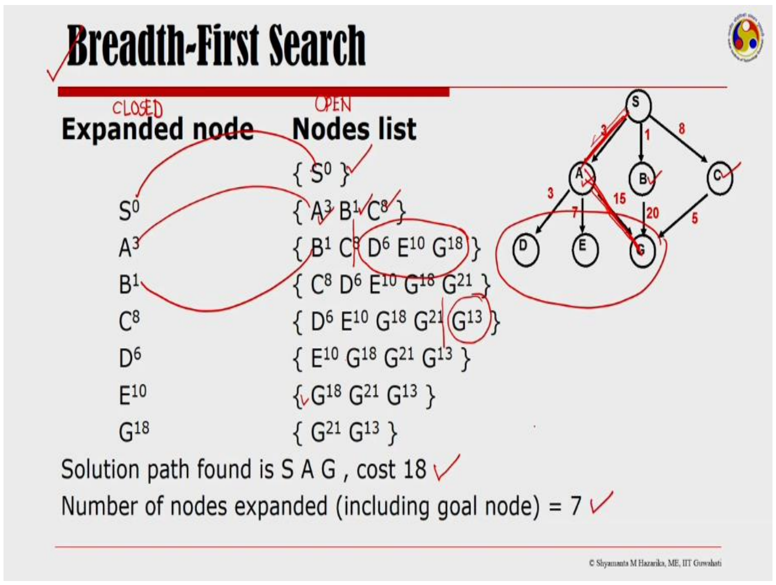


Illustrating Uninformed Search Strategies

© Shyamanta M Hazarika, ME, IIT Guwahati

So, let us now look back again and illustrate these uninformed search strategies that we have learnt today. So, we have we looked at breadth-first search, we have looked at depth-first search, we have looked at iterative deepening and we have looked at uniform-cost search. So, here is a small graph that I want all of you to look at. And over this we will try to explain and illustrate all of these search strategies. So, here is S; A, B, C its successors; D, E; then and B successor is G and C. So, I have put the cost of every path in red there besides each of the arcs.

(Refer Slide Time: 40:54)



Breadth-First Search

CLOSED Expanded node	OPEN Nodes list
	{ S^0 }
S^0	{ A^3 B^1 C^8 }
A^3	{ B^1 C^8 D^6 E^{10} G^{18} }
B^1	{ C^8 D^6 E^{10} G^{18} G^{21} }
C^8	{ D^6 E^{10} G^{18} G^{21} G^{13} }
D^6	{ E^{10} G^{18} G^{21} G^{13} }
E^{10}	{ G^{18} G^{21} G^{13} }
G^{18}	{ G^{21} G^{13} }

Solution path found is S A G , cost 18 ✓

Number of nodes expanded (including goal node) = 7 ✓

S
1
8
3
A
B
C
3
7
15
20
5
D
E
G

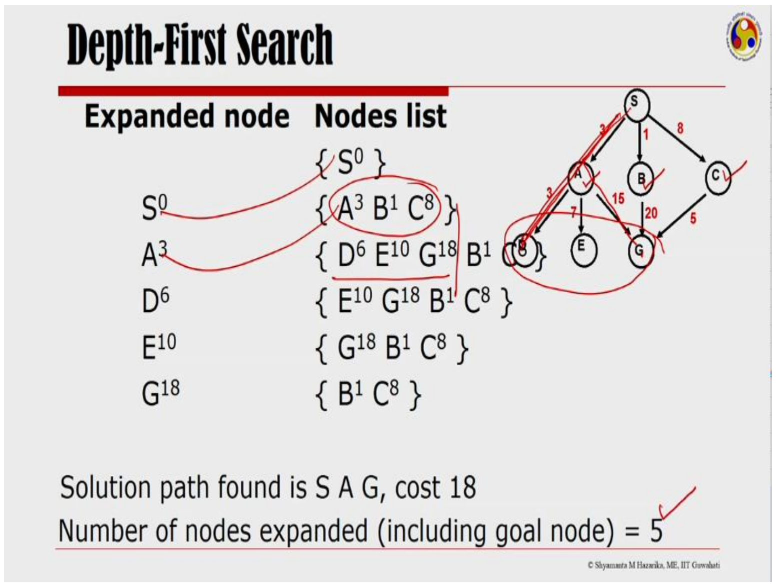
© Shyamanta M Hazarika, ME, IIT Guwahati

So, first let us look at breadth-first search. So, if you were looking at breadth-first search here, we know that we start with S 0. Here, S in that place. And then we generate its successors. So, we put this so in terms of the arcing that I was talking of. We call this node list is open and we call this node list closed. So, we create a list called closed and we create a list called open. In open, we have S 0 to start with.

And we pick up S 0, take S 0 here to close and get its successors into open list. So, I have successors A, B and C. So, I put A, B and C in that list. Then, the next node that I expand is of course A. And therefore, I move A here to this and I have now B, C. But, because it is a breadth-first search therefore I include its successors D, E and G to the end of the list. So, here is where D, E and G comes to the end of the list.

And the next node that is taken up for expansion is B. And its successors which is G is added to the end of the list here; so on and so forth. So, finally I have expanded E. And then, the next node that I get is this E. So, I have a solution because G is the goal node. So, the solution path that I am talking of is very simple S to A to G. So, that is the solution. And the cost of the solution is 3 + 15 = 18. Number of nodes expanded including the goal is that we could expand 1, 2, 3, 4, 5, 6, 7 nodes. So, we had expanded 7 nodes including the goal node.

(Refer Slide Time: 43:04)



Depth-First Search

Expanded node	Nodes list
	{ S^0 }
S^0	{ A^3 B^1 C^8 }
A^3	{ D^6 E^{10} G^{18} B^1 C^8 }
D^6	{ E^{10} G^{18} B^1 C^8 }
E^{10}	{ G^{18} B^1 C^8 }
G^{18}	{ B^1 C^8 }

Solution path found is S A G, cost 18

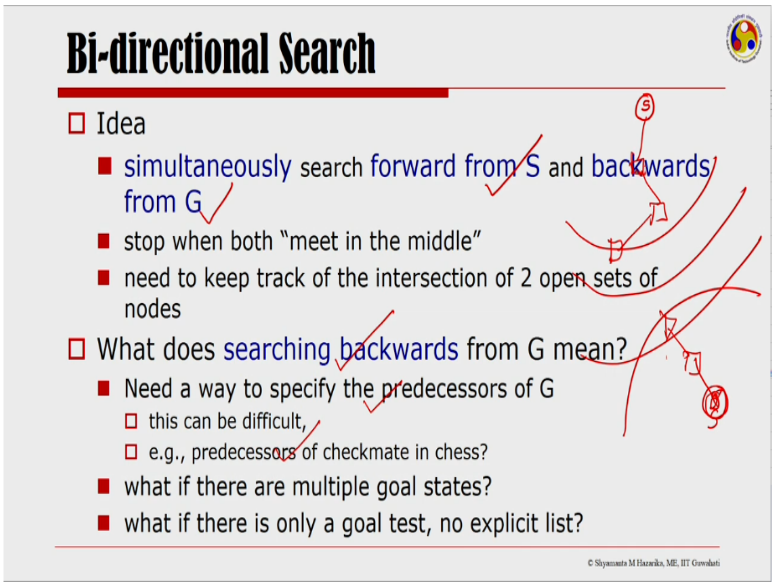
Number of nodes expanded (including goal node) = 5

© Shyamanta M Hazarika, ME, IIT Guwahati

The depth-first search is what we will try to expand now. So, in this, we take S, put it in this list. And next, we take the successors which are A, B and C and put it in the list called open. Take S 0 to close. Next, we take A 3 for expansion. Its successors D, E and G. Now, this is something that we need to take care of. We put it in the beginning of the list, here. So, as we put it in the beginning of the list, the fringe movement is along this line.

So, the next thing that comes to be expanded is D. And that is what, how it happens. And the solution path that I get here are also is S A G, the same solution path. But, what is more interesting is, because I took this path and then this was the maximum depth, I came back and expanded E again. And then I came back and got G. So, the number of nodes that I expand is only 5 including the goal node is only 5. This is depth-first search.

(Refer Slide Time: 44:32)



Bi-directional Search

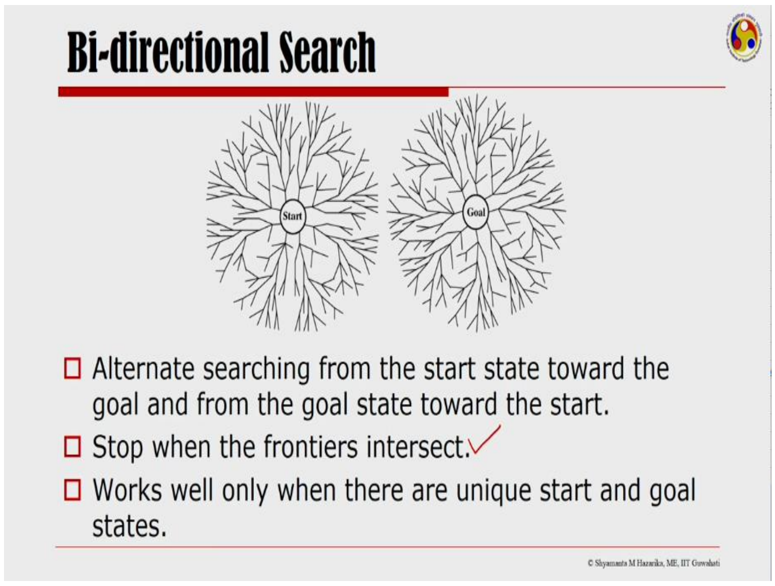
- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean?
  - Need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?

© Shyamanta M Hazarika, ME, IIT Guwahati

Now, we just highlight 1 idea of simultaneously searching from the forward from S and backward from G. So, stop when both meet. So, we have an node S here and we have a G here. We are looking for a solution from S to G. So, you keep on expanding and looking for a fringe. So, the fringe keeps on coming here. And from G you keep on going this side. And we stop when these 2 fringes come and meet.

So only point that needs to be now discussed or understood is, what does it mean to search from backwards form G. It basically means that we need to specify the predecessors of G. At times this could be difficult. Like in chess, what is the predecessor of checkmate. What if there are multiple goal states, there is a problem. What if there is only 1 goal test, no explicit list, so what do I do?

(Refer Slide Time: 45:37)

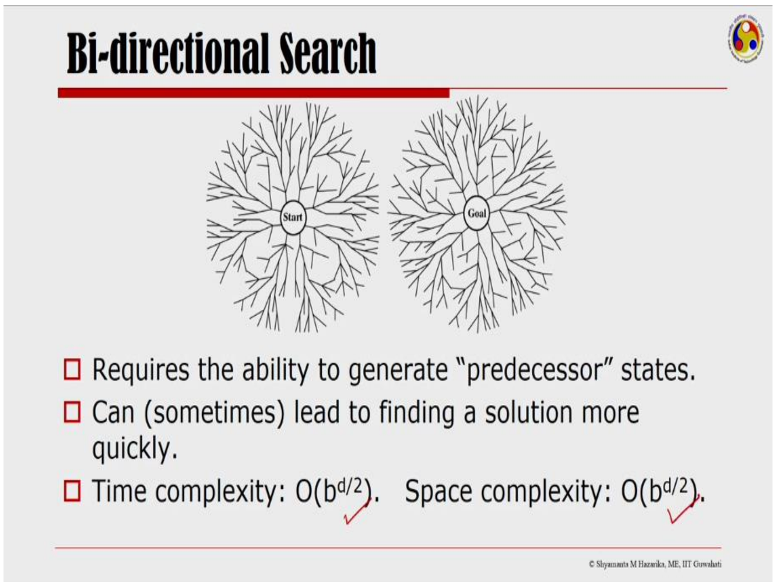


Bi-directional Search

- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.✓
- Works well only when there are unique start and goal states.

So, basically a bidirectional search as is an alternate searching from the start state towards the goal and from the goal towards the start. You stop when the frontiers intersect. Works well only when they are unique start and goal states.

(Refer Slide Time: 45:53)

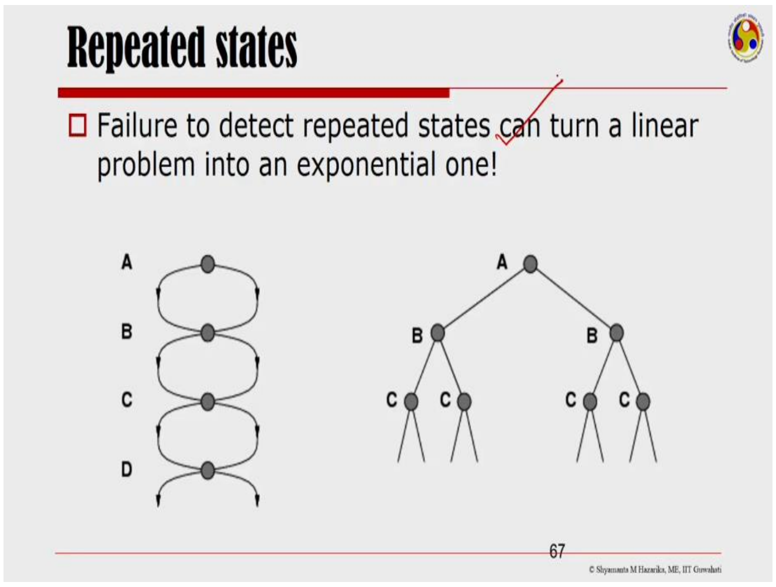


Bi-directional Search

- Requires the ability to generate "predecessor" states.
- Can (sometimes) lead to finding a solution more quickly.
- Time complexity: $O(b^{d/2})$.✓ Space complexity: $O(b^{d/2})$.✓

Requires the ability to generate predecessor states. That is something very important. And can sometimes lead to finding a solution more quickly. Here are the time and space complexities.

(Refer Slide Time: 46:07)



Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

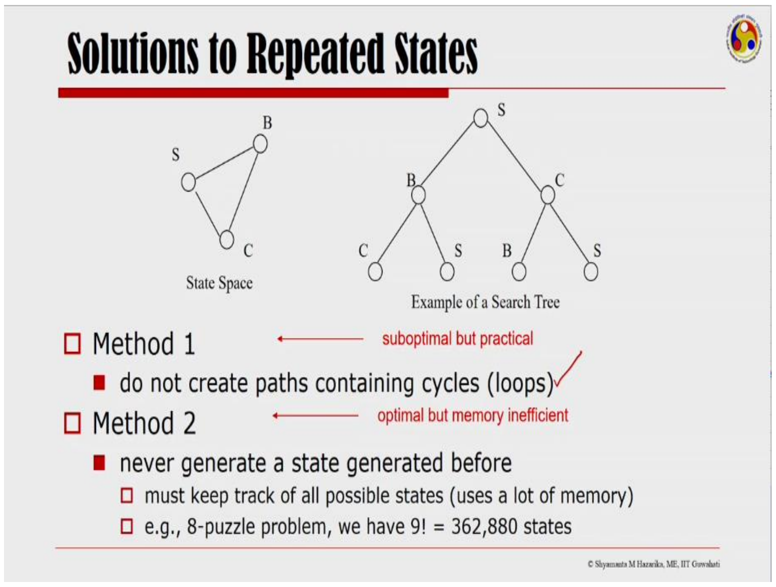
A
B
C
D
A
B
B
C
C
C
C

67

© Shyamanta M Hazarika, ME, IIT Guwahati

So, only thing that is important to realize when I am doing uninformed search over graphs for problem solving a state space search is failure to detect repeated states. Because, if I am, some failure to detect repeated states, this can turn a linear problem into an exponential one.

(Refer Slide Time: 46:31)



Solutions to Repeated States

S
B
C
State Space
S
B
C
C
S
B
S
Example of a Search Tree

- Method 1 ← suboptimal but practical
  - do not create paths containing cycles (loops)
- Method 2 ← optimal but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states

© Shyamanta M Hazarika, ME, IIT Guwahati

And there are 2 interesting solutions. In the first method which suboptimal but very practical, is I do not create paths containing cycles. This is what I highlighted in the beginning. And in the second method the idea is to never generate a state which is generated before. So, but then, this would require that you keep track of all possible states. So, it uses a lot of memory. For a very simple game like 8-puzzle, you would require something like factorial 9 states in such a case.

But then, and these solutions work very well because then you do not create exponential problems of the state space. So, this is all that we have in uninformed search. In the next class we will look at information from the problem domain being used to direct these searches; which are more commonly called the heuristic search techniques. Thank you.