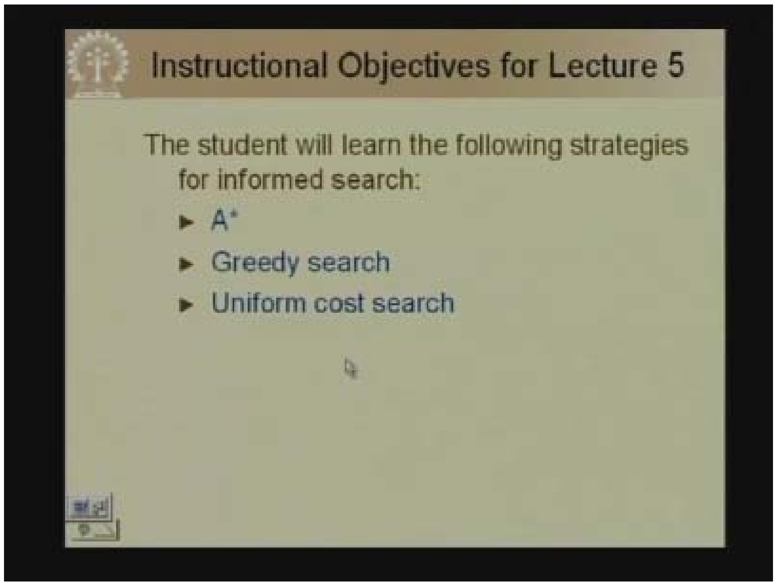
Artificial Intelligence

Prof. Sudeshna Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture - 5
Informed Search

Now we will star t the fifth lecture for this course Artificial Intelligence. Let us discuss about the module on search methods. Today we will primarily focus on informed search methods. In the last class we talked about several blind search strategies that do not use any problem specific information. We discussed depth first search, breadth first search as well as iterative deepening search. Today we are going to finish that discussion after talking about by bidirectional search and then we will move on to informed search that use heuristics information. In bidirectional search we will discuss the algorithm, the time and space complexities and then we will move onto informed search.

We will talk about the algorithm A star and before that we will talk about greedy search algorithm as well as uniform cost search which are special cases of A star.

(Refer Slide Time: 02:10)



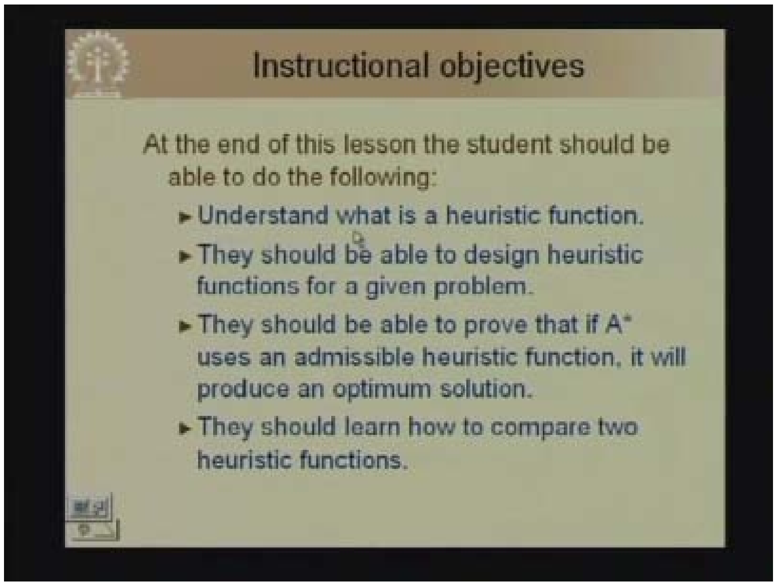
Instructional Objectives for Lecture 5

The student will learn the following strategies for informed search:

- A^*
- Greedy search
- Uniform cost search

At the end of this lesson the student should be able to do the following:

(Refer Slide Time: 02:21)



Instructional objectives

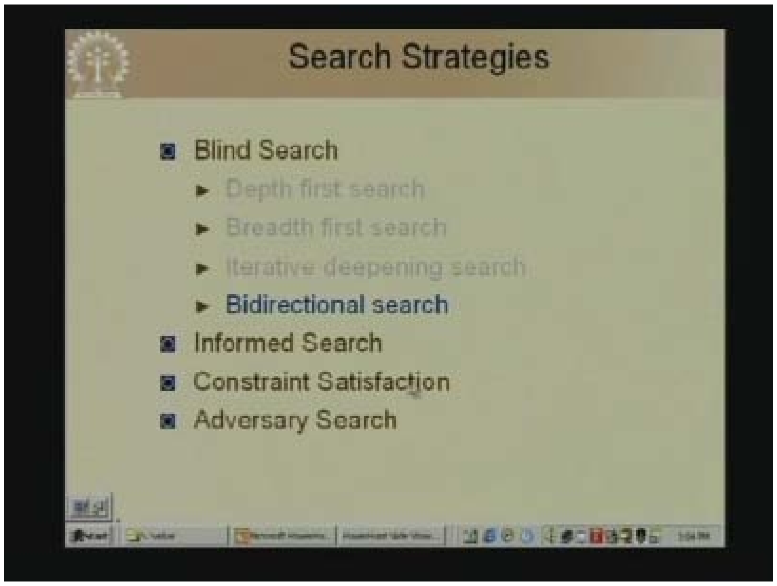
At the end of this lesson the student should be able to do the following:

- Understand what is a heuristic function.
- They should be able to design heuristic functions for a given problem.
- They should be able to prove that if A* uses an admissible heuristic function, it will produce an optimum solution.
- They should learn how to compare two heuristic functions.

He should be able to understand what a heuristic function is?
They should be able to design heuristic functions for a given problem. They should be able to prove that if A star uses an admissible heuristic function it will terminate and produce an optimum solution. They should learn how to compare two heuristic functions as well as how to combine multiple heuristics. So, in the last class we talked about blind search methods namely depth first search, breadth first search, iterative deepening search. Today we will talk about bidirectional search and then we will move on to informed search.

Subsequently in the other lectures we will talk about constant satisfaction which can be modeled as search problems and we will also look at adversary search which is used for dealing with two person games.

(Refer Slide Time: 03:23)

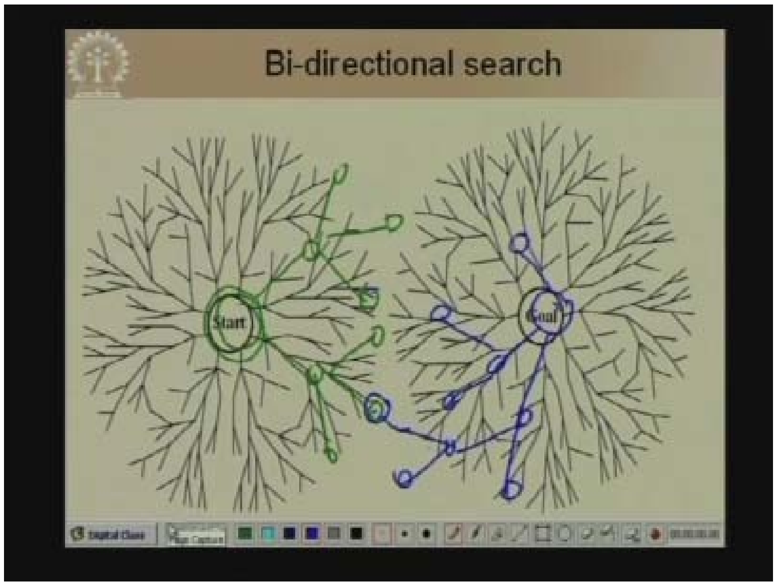


Search Strategies

- Blind Search
  - Depth first search
  - Breadth first search
  - Iterative deepening search
  - Bidirectional search
- Informed Search
- Constraint Satisfaction
- Adversary Search

Bidirectional search: In the other search methods we discussed one star ts from the star t node and then the search process explores the different nodes in search of a goal node. So the search branches out from the star t state. In bidirectional search, in addition one will also star t from a goal node and search backwards from the goal node trying to reach either the star t state or one of the states which is reachable from the star t state. So, if one can reach from a goal to a state which is also reachable from the star t state then we have found a path from the star t state to a goal state. The strategy which employs this is called bidirectional search.

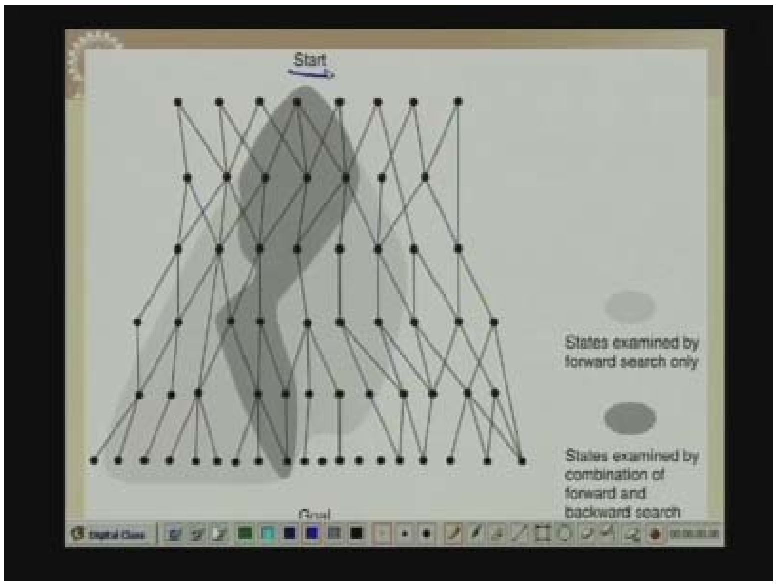
(Refer Slide Time: 04:40)



Bi-directional search

So we can look at this diagram which illustrates bidirectional search. This is the star t state and this is the tree which would be expanded if one star ts from the star t state and moves forward and one will be able to find the goal after examining these light grey states. In bidirectional search however, one would star t both from the star t state and examine certain portion of the star t state.

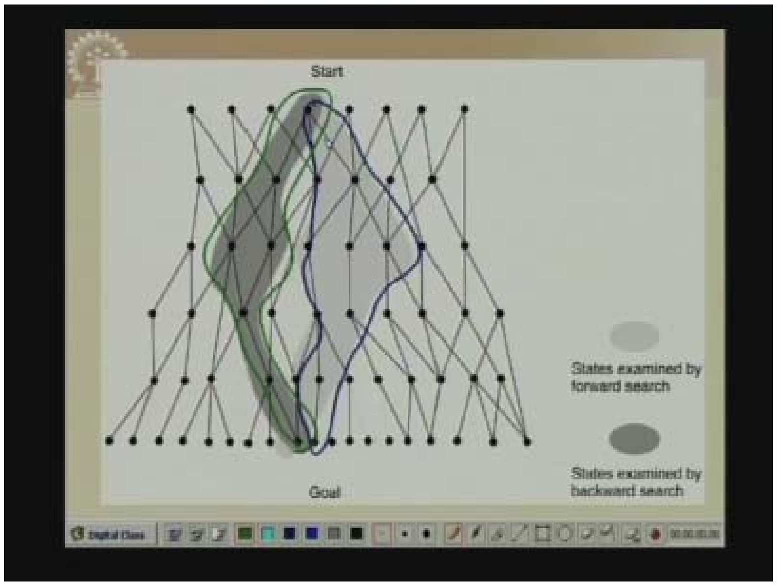
(Refer Slide Time: 04:53)


Start
States examined by forward search only
States examined by combination of forward and backward search
Goal

At the same time he will also star t from the goal state and move backwards until the search frontiers of the forward search as well as the backward search meet. And if they meet at a particular node we would be able to find a path from the path to the goal state. So we see that it is possible that in bidirectional search one many to expand fewer nodes than one would if one carried on forward search.

However, this may not always be the case. This is an example where the forward search is illustrated by this light grey envelope. So this is the envelope of forward search and this is the envelope of backward search. And we see that these two envelopes do not really meet so that these paths are disjoined so we do not save on expanding any nodes if we do bidirectional search.

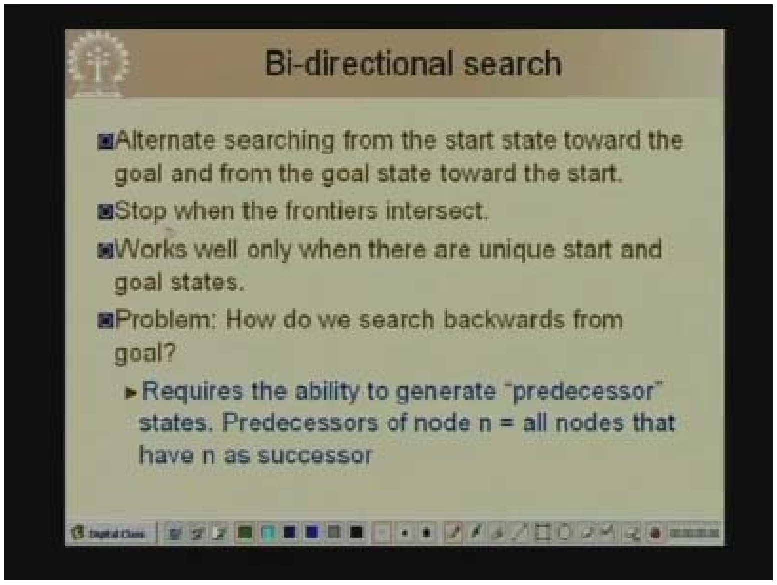
(Refer Slide Time: 06:48)


Start
Goal
States examined by forward search
States examined by backward search

In bidirectional search what we do is that we carry on the search process forwards from the star t state as well as backwards from the goal and we alternate these two phases.

For example, we first expand some nodes in the forward direction and then expand some nodes in the backward direction then again in the forward direction again in the backward direction and so on. Every time we expand a node we need to check whether that node has been expanded in the other search tree. If it has been expanded then we would have found a path from the star t to the goal. That is, we stop when the frontiers of the forward and backward tree intersect.

(Refer Slide Time: 07:41)



Bi-directional search

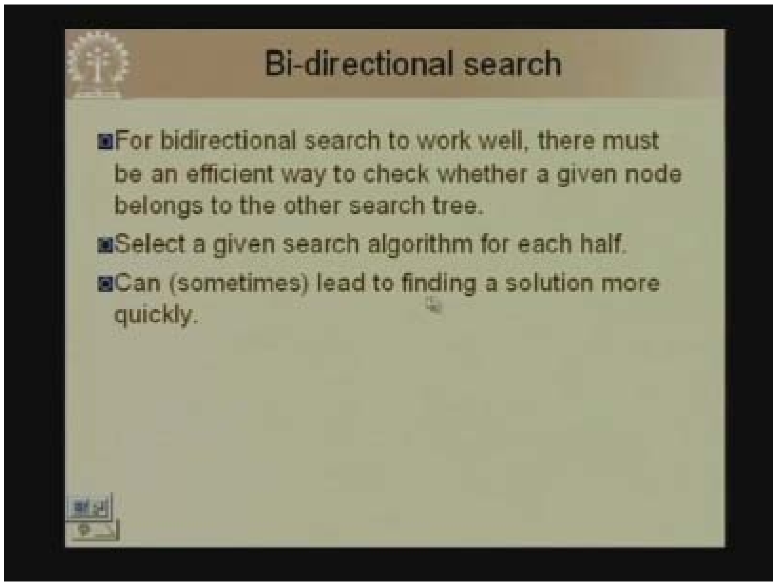
- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states.
- Problem: How do we search backwards from goal?
  - Requires the ability to generate "predecessor" states. Predecessors of node n = all nodes that have n as successor

However, in order to do the bidirectional search we have to star t from a goal state. There are many problems where there are many goal states. It is difficult to decide which goal state we should star t from. Therefore bidirectional search works best if there is one goal state for the problem which is easy to get to. The second problem one might encounter in bidirectional search is, how do we search backwards?
For all problems it may not be possible to search backwards. So, being able to search backwards means we have these reversible operators. That is, we can generate the predecessors of a state as well as the successors. In such problems bidirectional search is helpful.

Also, we alternate from searching in the forwards direction and searching in the backwards direction and every time we expand a node we have to check whether that node occurs in the frontier of the other search tree. Therefore for bidirectional search to work well we would need an efficient way to check whether a given node has already been expanded.

If we take every node in the frontier and check whether it is the same as the correct node then any advantage we get by bidirectional search will easily be lost. And then for each of the forward search as well as the backward search we have to select a given search algorithm. However, bidirectional search can sometimes lead to finding a solution more quickly.

(Refer Slide Time: 09:50)



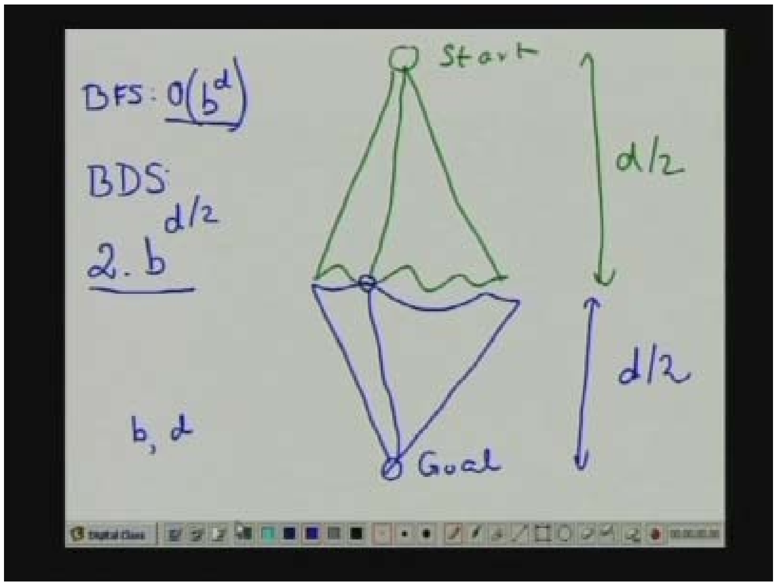
Bi-directional search

- For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
- Select a given search algorithm for each half.
- Can (sometimes) lead to finding a solution more quickly.

So let us see what we would gain if we use bidirectional search. Suppose this is our start state and this is our goal state and let us say the branching factor is b and the distance from star t to goal is d then breadth first search for example would expand order BFS will expand order b power d nodes and that would be the time and space complexity. In bidirectional search if we are lucky the forward tree and the backward tree will meet exactly half way. So this is the forward tree and this is the backward tree and if we are lucky we have found an intersection of the forward tree and the backward tree. Therefore we would get a path from the goal to this node as well as from the star t to this node that is we will have a path from star t to goal.

Now in the best case this distance will be d by 2 and also this distance will be d by 2. So the number of nodes expanded in bidirectional search would be two times b power d by 2 which is better than b to the power d nodes expanded by BFS. So, in bidirectional search in the best case we will get two times b power d by 2 nodes. However, the space complexity is also b power d by 2 because we would have to store the frontier of at least one of the search trees. Like often what is done is we do breadth first search in one direction and DFS in the other direction. So at least for one of the search trees we must have a breadth first search type of procedure where we have a frontier which can be ordered b power d by 2.

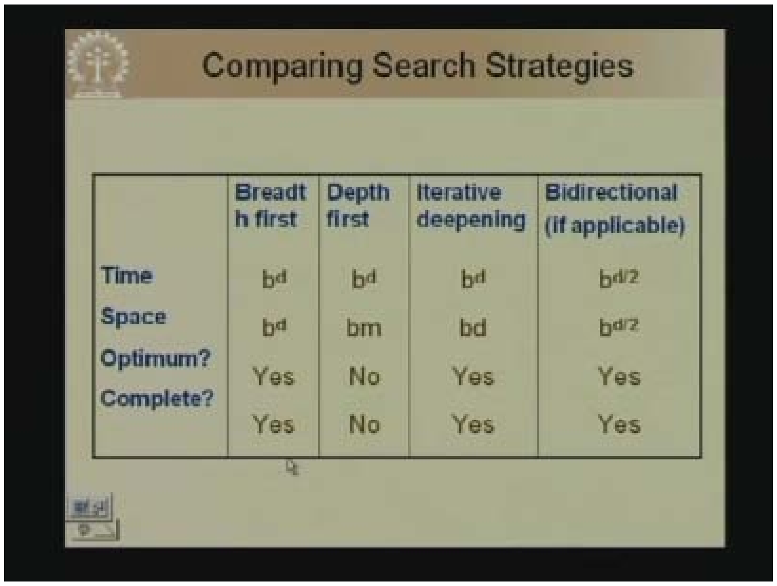
(Refer Slide Time: 12:35)



Let us just go back and compare the different line search methods we have considered so far. Breadth first search requires time of b power d space of b power d it is optimum and it is complete. Depth first search requires time of order b power d space of bm where m is the depth of the search tree, it is not optimum and it is not complete. Iterative deepening search has a time complexity of b power d and space complexity of only b into d where d is the length of the cheapest solution, it is optimum, it is complete.

Bidirectional search when it is applicable might have a time complexity of b power d by 2, space complexity of b power d by 2 it is optimum you can show that and it is also complete. However, in some cases unless you have a very efficient way of checking with the frontier of the other search tree they could be more overhead involved in trying to check if a node is there in the search tree.

(Refer Slide Time: 14:14)

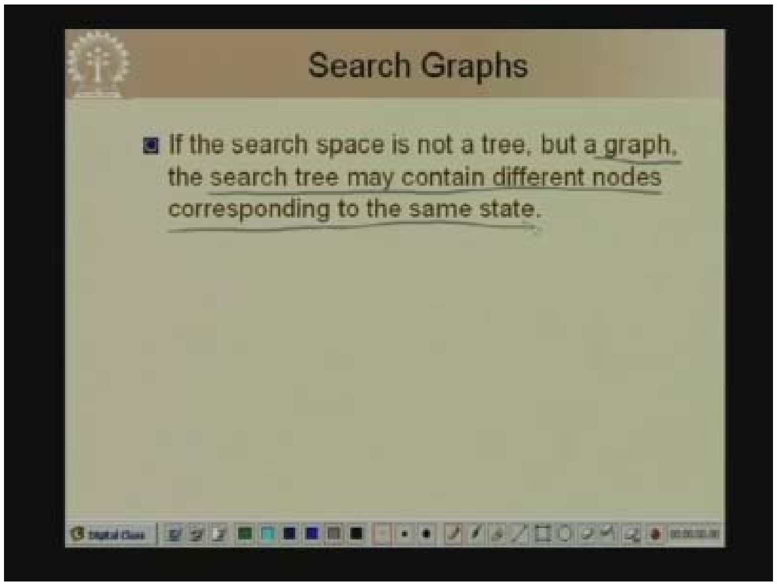


Comparing Search Strategies

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

Now, we have so far considered search trees where we star t from the search star t state and unfold the tree we get. Now it is quite possible that the search space is more like a graph but not a tree. That is, from the star t state to a node n there could be multiple paths. If we look at the search space as a tree we might be getting to that node many times over and we are expanding a node more than once. So, if you want to deal with this we must consider that the search space may be graph and that in this case the search tree may contain different nodes corresponding to the same state. These are examples of some search spaces which contain nodes more than once.

(Refer Slide Time: 15:12)

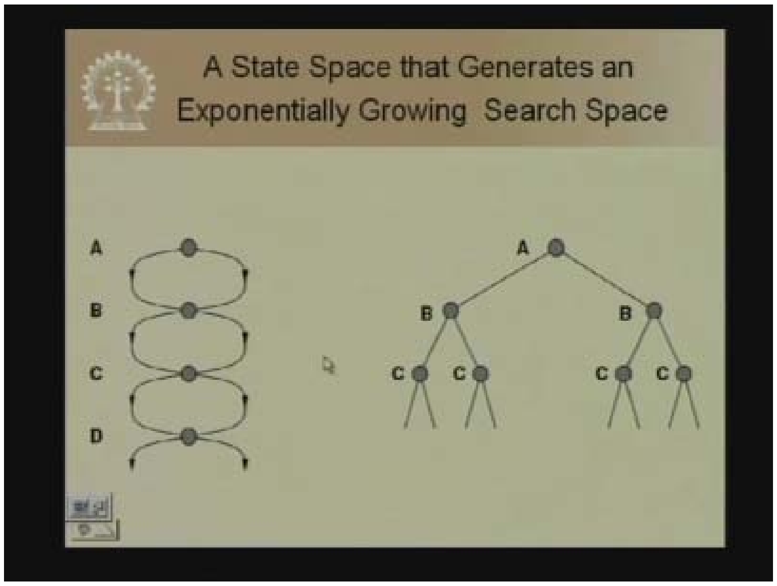


Search Graphs

- If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state.

Look at this search graph:
This is node a, this is node b, this is node c, this is node d and so on. There are two paths, two arcs from A to B, two arcs from B to C, two arcs from C to D and so on. Now if we unfold this as a tree this is A, this is B, this is C C C C. So there would be 8 nodes corresponding to D, 16 corresponding to E and so on. So if you have n nodes this tree will have size of the order of 2 power n so the search tree can be exponentially larger than the search space.

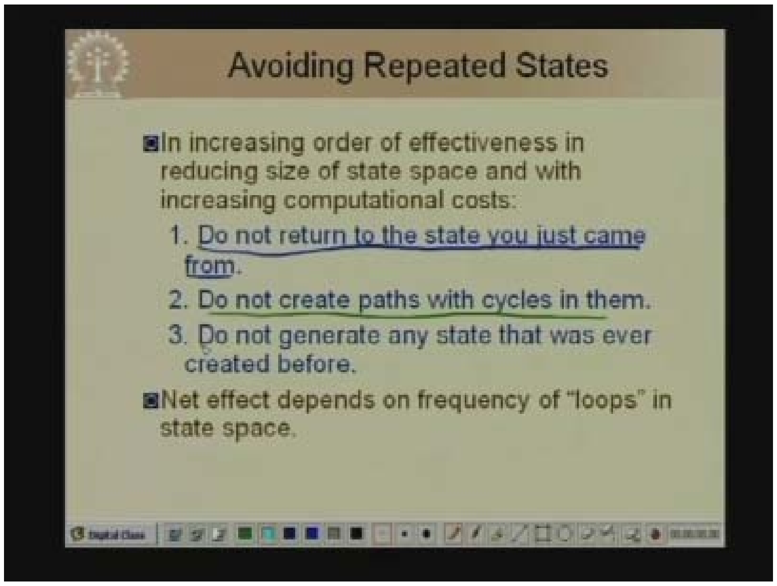
(Refer Slide Time: 16:21)



A State Space that Generates an Exponentially Growing Search Space

Tree search methods are not very good for such types of search spaces. So, in order to have an algorithm which can work efficiently when the search space is a graph we need to avoid repeated state. There are different mechanisms we could use. For example, we can choose not to return to the state we just came from. This is a very simple trick that we often use in 8 puzzle. When we expand a node expand a node and generate its successors we do not generate its immediate parent. By this we mean we can avoid some sort of duplication. A more sophisticated method is not to create paths with cycles in them. That is, when we expand a node to generate its successor we check that the successor does not occur in the path from the node to the root. That is, we do not generate in cycles. And the best way of doing this is not to ever generate a node more than once by checking whenever we generate a state whether it was ever created before.

(Refer Slide Time: 17:51)

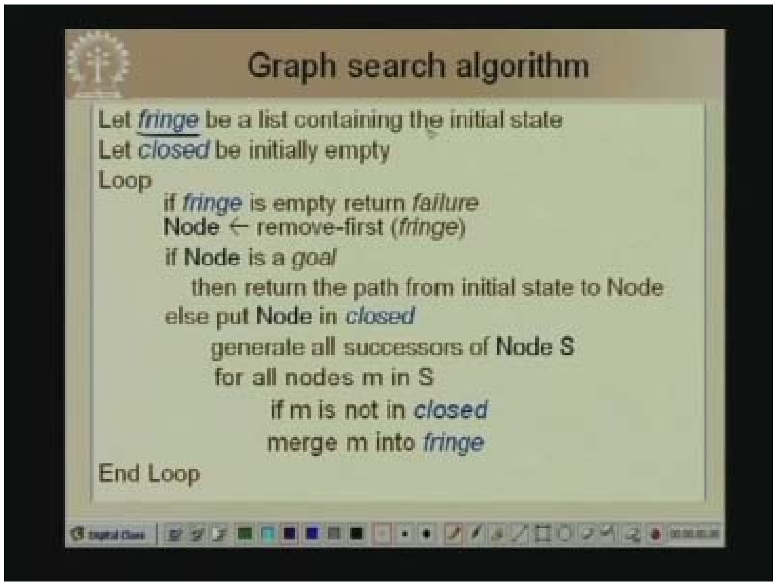


Avoiding Repeated States

- In increasing order of effectiveness in reducing size of state space and with increasing computational costs:
  1. Do not return to the state you just came from.
  2. Do not create paths with cycles in them.
  3. Do not generate any state that was ever created before.
- Net effect depends on frequency of "loops" in state space.

Unfortunately for doing this we must keep track of all the nodes that were expanded ever and we must check that this node was not expanded before. We need to maintain a list other than the fringe in the search algorithms we have. We usually call this particular list closed. So, closed is a list which keeps track of the entire expanded node and whenever we generate a node we check whether or not it is already enclosed. This is the basic graph search algorithm which is a variation of the basic tree search algorithm that we looked at earlier.

(Refer Slide Time: 18:43)



Graph search algorithm

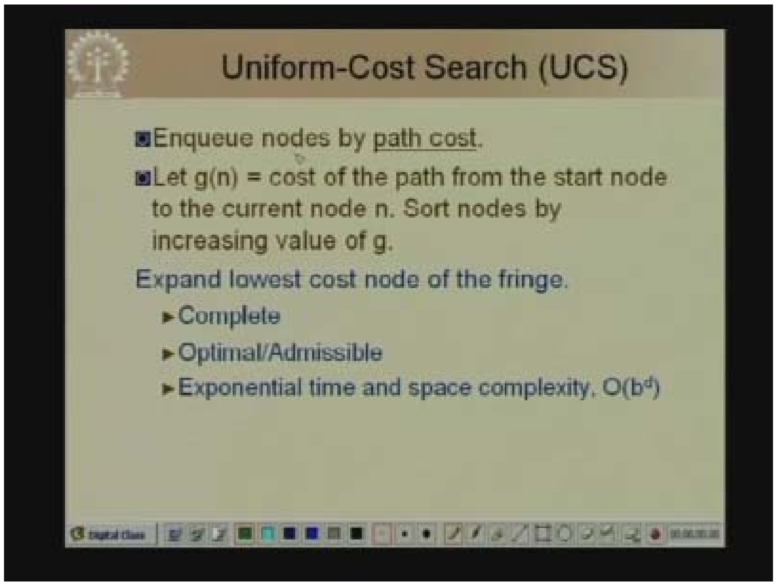
```
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
              generate all successors of Node S
              for all nodes m in S
                      if m is not in closed
                      merge m into fringe
End Loop
```

So fringe is the list containing the initial state. A fringe is often referred to as open. So these are the nodes which are in the frontier of the search tree and which are candidates for expansion. It is called the open list or the fringe. Closed is another list which is initially empty and it will keep all the expanded nodes. So the algorithm proceeds like this: We have a loop. If fringe is empty that is no more nods to expand we return failure. Otherwise we remove the first node from fringe and check that node is a goal or not.

If node is a goal we have found a goal state and we return by tracing the pointers and finding the path from the root to this node. Otherwise if node is not a goal node we put node in closed. Generate all successors of node. Let us call them S. S is the set of successors of node. For all nodes m in S if m is going to be generated if m is not in closed then you add m to fringe but if m is in closed you ignore it. Therefore we have modified the tree search algorithm for graph search. The basic difference is we have included this closed list which was initially empty. Whenever we expand a node we put it closed. When we generate a new node we check whether it is already in closed. If it is closed then we do not generate it else we generate it. Now let us look at a variation of breadth first search which we call uniform cost search.

As we discussed breadth first search expands node according to its level, level by level. So it expands nodes which are smaller number of steps away from the star t earlier. However, sometimes arc costs are not uniform. So, instead of generating nodes level by level we would like to generate those nodes that have smaller cost from the parent. So we might like to generate nodes in the order of their distance from the parent and this is done by Uniform Cost Search UCS.

(Refer Slide Time: 21:31)



Uniform-Cost Search (UCS)

- Enqueue nodes by path cost.
- Let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.

Expand lowest cost node of the fringe.

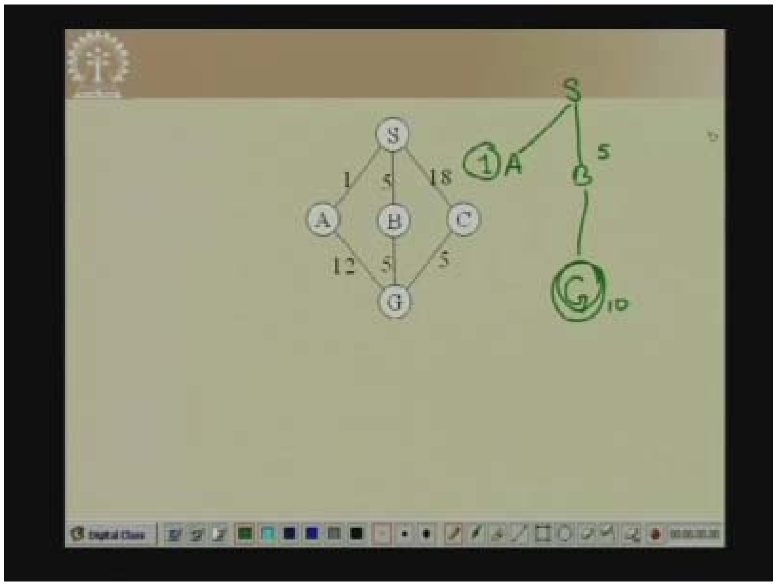
- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

In Uniform Cost Search when we put the nodes in the queue the order of the nodes by the path cost from the root to that node. But instead in breadth first search what we do is we maintain a queue for storing the fringe. That is, the nodes expanded are added at the back of the queue. In Uniform Cost Search we maintain the fringe as a priority queue. The priority of a node is its distance from the root. So, nodes are enqueued by the path cost for which a priority queue can be used. The heap data structure is ideal for storing a priority queue.

We denote by g(n) the cost of the path from the star t node to the current node n. So g(n) is the cost of the path from the star t node to the current node. We store nodes in the priority queue according to the value of g. And the algorithm expands the lowest cost node of the fringe. The algorithm can be shown to be complete. It is optimal or admissible and it has exponential time and space complexity in the worst case. As an example let us look at this search graph consisting of these five nodes.

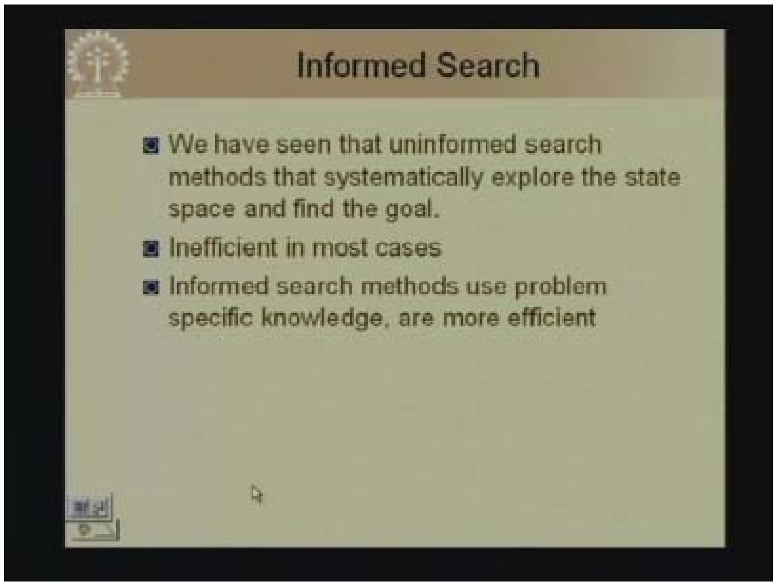
This (Refer Slide Time:: 23:04) is the star t node and this is the goal node. In Uniform Cost Search we will star t will the star t state s. First generate A which has a g value of 1 then generate B which has a g value of 5. Then the candidates for expansion are g along this path with a g value of 13, g along this path with g value of 10, c along this path with a g value of 18. This is smaller so we will expand this g which has a g value of 10. And then we will try to find out which is the shortest path and then we will expand this node goal and we would have found the shortest cost path from the star t state to the goal state.

(Refer Slide Time: 23:58)



As a result of breadth first search we can find the smallest length path from the star t to the goal. If you do Uniform Cost Search you can find the minimum cost path from the star t to the goal. Next we will come to Informed Search.

(Refer Slide Time: 24:27)



Informed Search

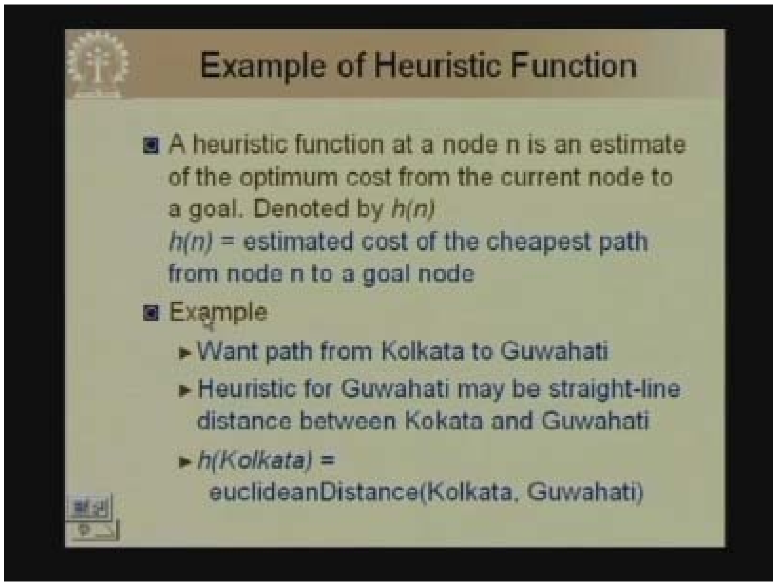
- We have seen that uninformed search methods that systematically explore the state space and find the goal.
- Inefficient in most cases
- Informed search methods use problem specific knowledge, are more efficient

In Informed Search we use heuristics about the problem domain. Uninformed search methods we looked at earlier systematically explore the state space to find the goal. They are not very efficient in most cases. We saw that most of the time the complexity of the algorithm was order b power d that is exponential in the length of the search path. Informed Search method tries to improve problem solving efficiency by using problem specific knowledge.

Let us first try to see what we mean by heuristics. Heuristics literally means rule of thumb. This is a definition of heuristics by Judea pearl. Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. In Informed Search we use heuristics to identify the most promising search path.

Let us look at some examples of heuristic function. A heuristic function at a node n which we will use for the purpose of the search algorithms is an estimate. The heuristic function is an estimate of the optimum cost from the current node to a goal. We usually denote a heuristic function at a node n by h(n) which is the estimated cost of the cheapest path from node n to the goal node.

(Refer Slide Time: 25:48)

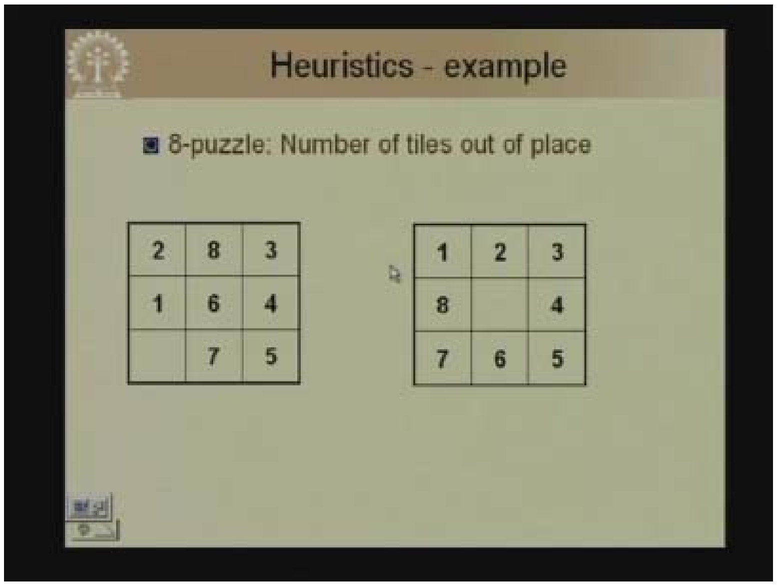


Example of Heuristic Function

- A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. Denoted by *h(n)*
  h(n) = estimated cost of the cheapest path from node n to a goal node
- Example
  - Want path from Kolkata to Guwahati
  - Heuristic for Guwahati may be straight-line distance between Kokata and Guwahati
  - *h(Kolkata)* = euclideanDistance(Kolkata, Guwahati)

For example, let us say we want to get a path from Kolkata to Guwahati, so here we have a map. This is Kolkata on the map and this is Guwahati on the map. And the actual path from Kolkata to Guwahati might be this. If you do not know what the actual path is, an estimate of the distance of this path is the Euclidean distance or the shortest distance between Kolkata and Guwahati. So this distance, the straight line distance between k and g is an underestimate of the actual distance of the path from k to g.

(Refer Slide Time: 27:11)



Heuristics - example

- 8-puzzle: Number of tiles out of place

2	8	3
1	6	4
	7	5

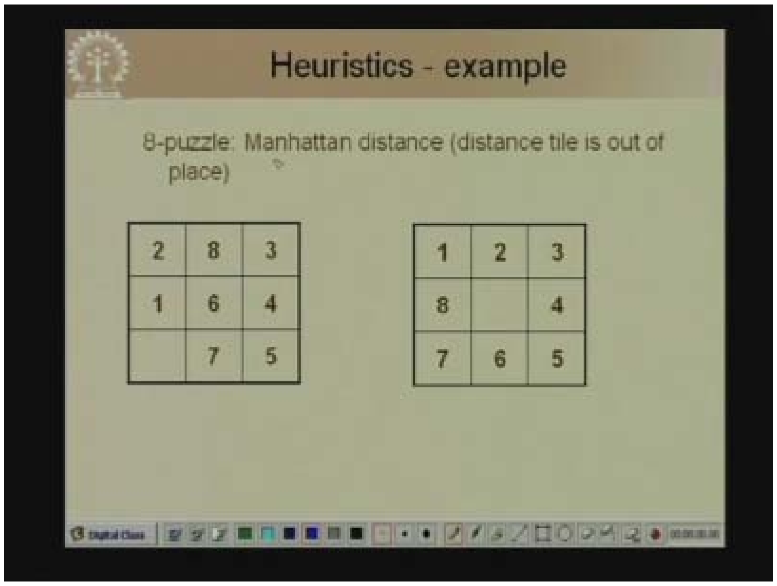
1	2	3
8		4
7	6	5

Let us look at the game of 15 puzzle we discussed earlier. This is one configuration for the 8 puzzle. So this is a given configuration of 8 puzzle and this is the goal we are trying to achieve. One heuristic for 8 puzzle is the number of tiles out of place.

What is the actual cost to get from n to goal?
We have to take a number of move to get from n to goal and that is what we would like to find out. However, it is not easy to look at this and say what would be minimum cost path. But what we can easily do is find out the number of tiles which are not in their correct location.

For example, look at 2 here, 2 is not in its correct location in state n. In order to move to its correct location we have to move 2 at least once. Similarly, 8 is not in its correct location, 3 is in its correct location, 1 is not in its correct location, 6 is not in its correct location, 4 is in the correct location, 5 is in the correct location, 7 is not in the correct location. So the heuristic at node n is equal to 5 because five tiles are not in their correct location. And we must make at least five moves to move them to their correct location. So h(n) that is 5 is an underestimate of the actual number of steps required to move to the goal state.

(Refer Slide Time: 29:10)



Heuristics - example

8-puzzle: Manhattan distance (distance tile is out of place)

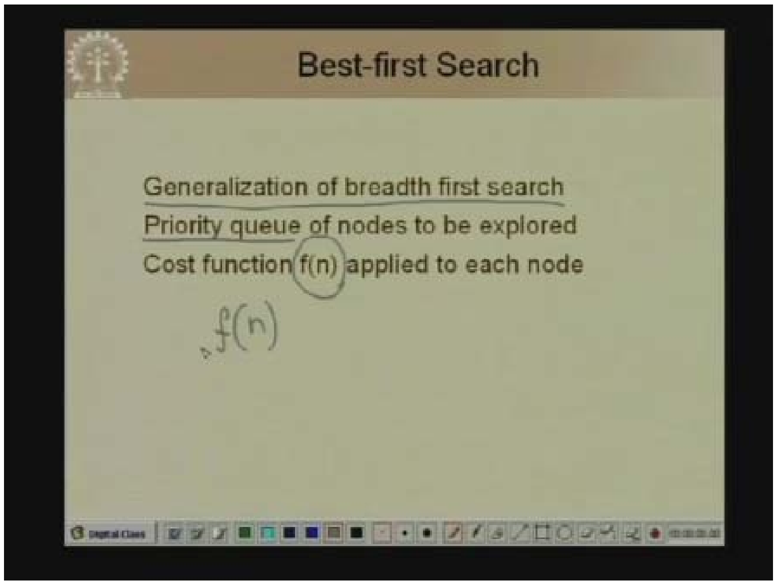
2	8	3
1	6	4
	7	5

1	2	3
8		4
7	6	5

Another heuristics for 8 puzzle is the Manhattan distance heuristic. For example, let us say 2 this node 2 is not in its correct position. In order to move to its correct position we have to move to one position to the right. However, look at node 8. Node 8 is not in its correct position. It has to move to this position. To move node 8 to this position we have to move one step down and one step to the left. That is, we need at least two moves. To move 6 to its correct position we need at least one move. To move 7 to its correct position we need at least one move. To move 1 to its correct position we need at least one move. So h(n) in this case is, one move for 2, two moves for 8, one move for 6, one move for 7 and one move for 1 that is in this case it is equal to 6. Therefore this is an underestimate of the actual number of moves required to move from this state to this state.

Now we will look at another search algorithm which uses this heuristic information. Best first search is a generalization of breadth first search where the fringe or the open list is maintained as a priority queue and a cost function f(n) is used which denotes the priority of a node. So f(n) is the cost function of the node which denotes the priority of the node.

(Refer Slide Time: 31:20)



Best-first Search

Generalization of breadth first search

Priority queue of nodes to be explored

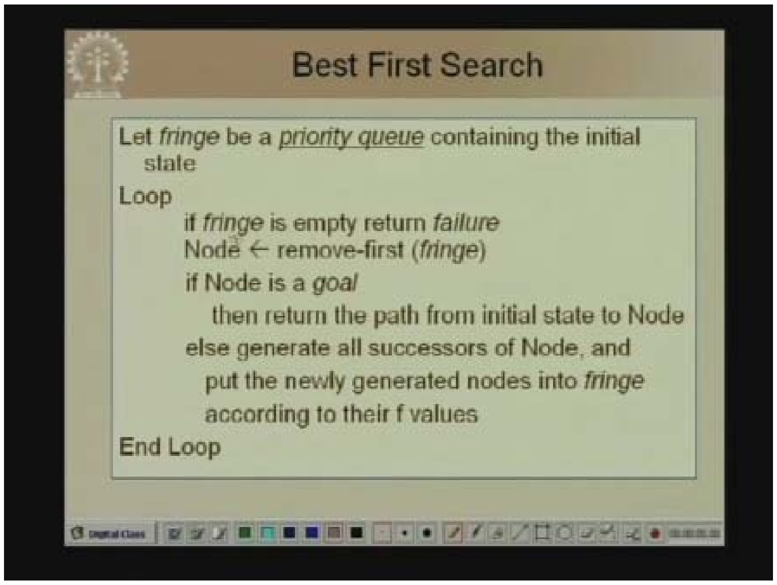
Cost function f(n) applied to each node

$f(n)$

Nodes are put in fringe sorted according to the value of f(n). So this is the modification of the basic search algorithm using the function f(n). Fringe is maintained as a priority queue containing the initial state.

If fringe is empty return failure otherwise remove the element with the highest priority from fringe let that be node, if node is goal we return the path from initial state to the node otherwise we generate all successors of node. Put the newly generated nodes into fringe according to their f values, end loop. So best first search is a variation of the search algorithm where fringe is maintained as a priority queue and nodes are put in the priority queue ordered by their priorities which is denoted by the f value of a node.

(Refer Slide Time: 31:41)

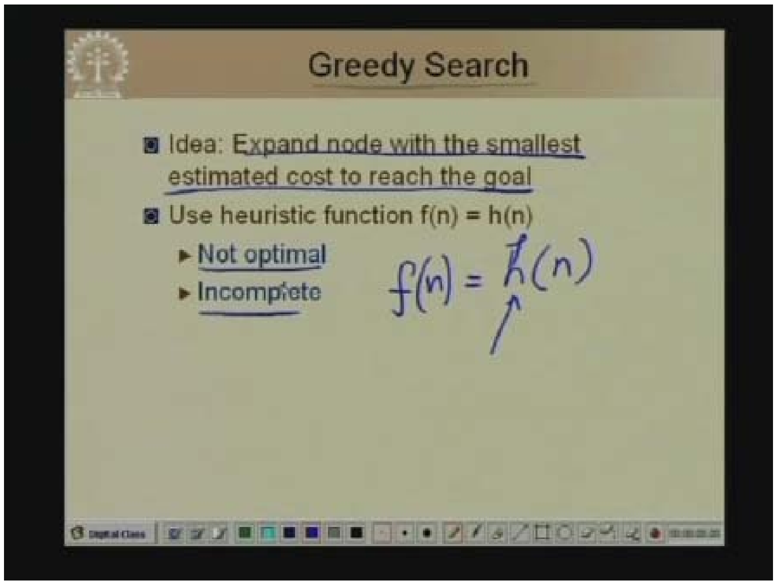


Best First Search

```
Let fringe be a priority queue containing the initial
   state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
        if Node is a goal
           then return the path from initial state to Node
        else generate all successors of Node, and
          put the newly generated nodes into fringe
          according to their f values
End Loop
```

Let us try to now look at different variations of best first search according to the function f that they use. The simplest algorithm we will look at is the Greedy Search algorithm. In Greedy Search we always expand the node with the smallest estimated cost to reach the goal. That is, the f value of a node is h value of a node. So h is a heuristic function. The h value of a node is an estimate of the actual path cost from the nod to reach the goal. So we use f(n) is equal to h(n). However, we can show that this search algorithm is not optimum and is not complete.

(Refer Slide Time: 33:20)



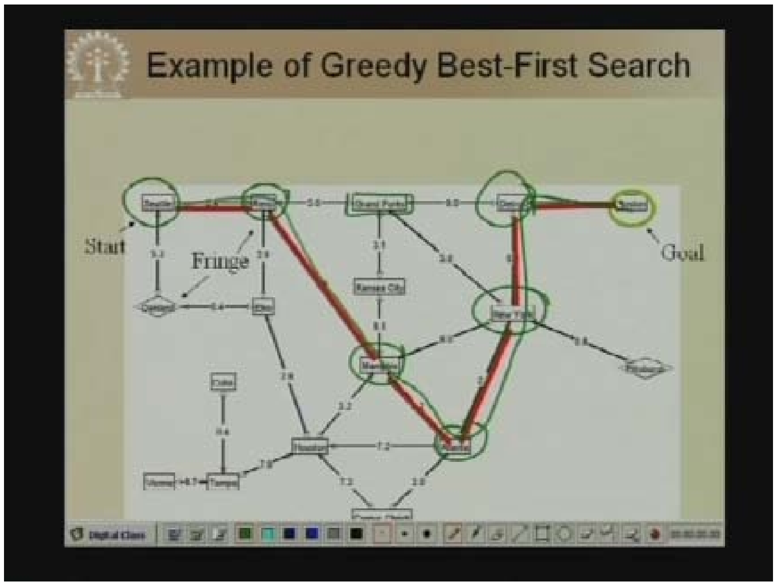
Greedy Search

- Idea: Expand node with the smallest estimated cost to reach the goal
- Use heuristic function f(n) = h(n)
  - Not optimal
  - Incomplete

$$f(n) = h(n)$$

This is an example of greedy method. We are star ting here from the star t state which is Seattle and our objective is to reach Boston. So these are the arcs, these rectangles are the cities and the r x denote the cost between cities. Now, if we do greedy search we will evaluate a heuristic function at each of these nodes and from Seattle we will move to that city which has the smallest h value to Boston. and in this case suppose if it turns out to be Reno then from Reno we go to Memphis, Memphis to Atlanta to New York to Detroit and to Boston and we get a path like this. However, greedy search may not always give us the optimum solution which we can show.

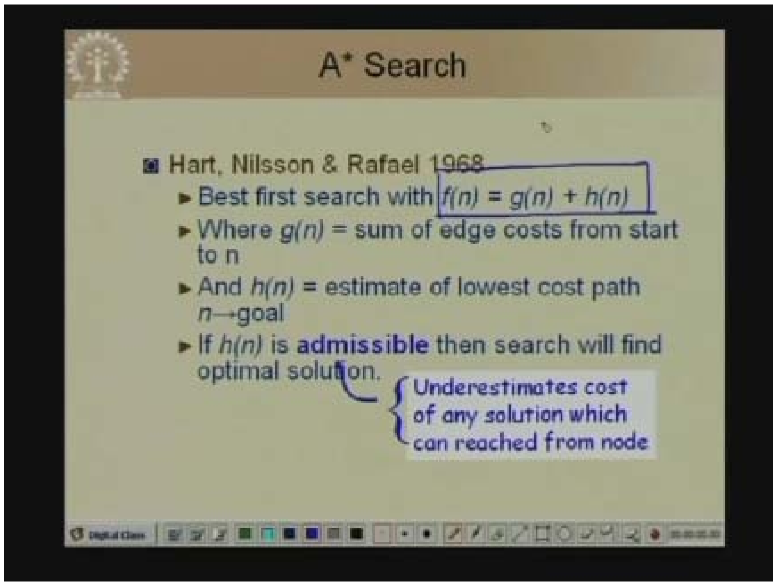
(Refer Slide Time: 34:27)



Example of Greedy Best-First Search

Next we will come to A star search which uses a slightly more sophisticated heuristic function. So in best first search f(n) is equal to g(n) plus h(n).

(Refer Slide Time: 34:55)



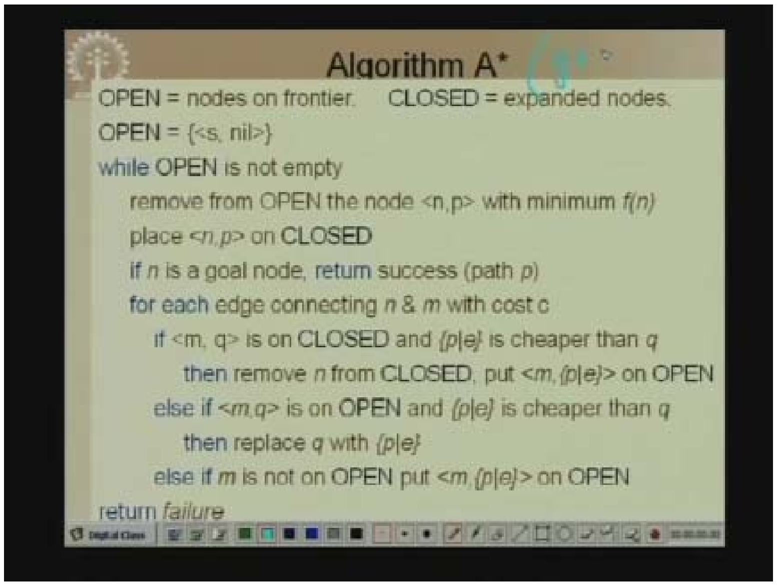
A* Search

- Hart, Nilsson & Rafael 1968
  - Best first search with $\boxed{f(n) = g(n) + h(n)}$
  - Where $g(n)$ = sum of edge costs from start to n
  - And $h(n)$ = estimate of lowest cost path $n \rightarrow$ goal
  - If $h(n)$ is **admissible** then search will find optimal solution.

Underestimates cost of any solution which can reached from node

We have seen that g(n) is the cost to get to the node from the star t state and h(n) is an estimate to get from the node n to the goal state. That is, g(n) is the sum of the edge costs from star t to end and h(n) is an estimate of lowest cost path from n to goal. Now it is a theorem and we will show that if h(n) is admissible then A star search which uses f(n) is equal to g(n) plus h(n) as the priority function which will find a optimum solution. And h(n) is admissible if it underestimates the cost of any solution which can be reached from node. So an underestimating heuristic function is an admissible heuristic function which gives an estimate which is a lower bound of the actual cost. Now this is the algorithm A star for graphs.

(Refer Slide Time: 36:16)



Algorithm A*

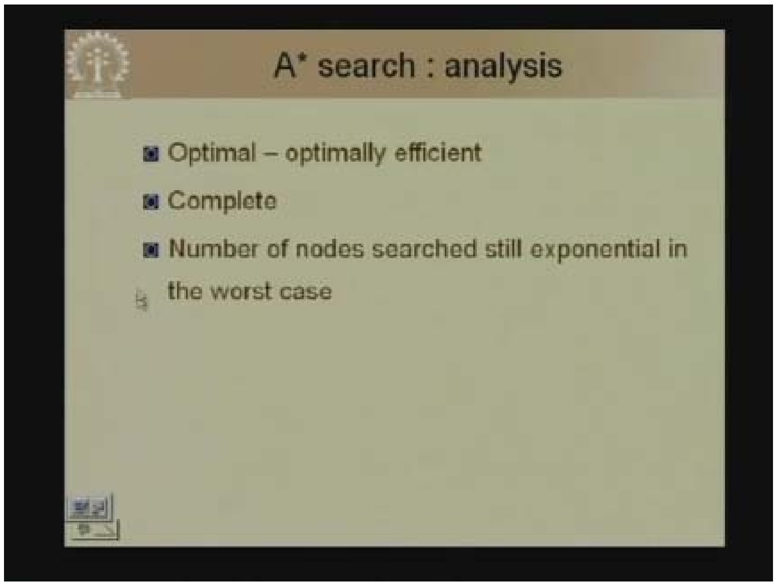
```
OPEN = nodes on frontier.    CLOSED = expanded nodes.
OPEN = {<s, nil>}
while OPEN is not empty
    remove from OPEN the node <n,p> with minimum $f(n)$
    place <n,p> on CLOSED
    if n is a goal node, return success (path p)
    for each edge connecting n & m with cost c
        if <m, q> is on CLOSED and {p|e} is cheaper than q
            then remove n from CLOSED, put <m,{p|e}> on OPEN
        else if <m,q> is on OPEN and {p|e} is cheaper than q
            then replace q with {p|e}
        else if m is not on OPEN put <m,{p|e}> on OPEN
return failure
```

This is a generalized A star called graph search. Here we maintain two lists open and closed. Open maintains the nodes on the frontier of the search tree and closed maintains the expanded nodes. Initially we put the start sets on open. We also keep at every node a pointer to its path from the root. That is, we keep a pointer to its parent. So open contains the tuple s nil. Then we have a loop, while open is not empty we remove from open the node n, p which has the minimum value of f(n). We put n, p on closed.

If n is a goal node we return success and we return the path p for each edge connecting n and m with cost c. So n is the node that we are expanding and we find its successors. For each successor m of node n we check if m is in closed with a path q with the parent point q. So, if m q is on closed already and the current path the path from n is p. So the current path for m is p concatenated with e. Therefore if the cost of the path p, e is cheaper than the path q then the current path to m is cheaper than the path we have already got so we will consider this path.

However, otherwise the path we obtained earlier for m was cheaper and we will throw away the current node m. So if m, q is already on closed and p concat e is cheaper than q we remove n from closed and put m, p concat e on open. Otherwise if m, q is not on closed but m, q is on open and p concat e is cheaper than q we replace q with p concat e. Otherwise if m is not on open we put m and the path p concat e on open and when open is empty we return failure. So this is the generalized algorithm for A star search.

(Refer Slide Time: 39:13)

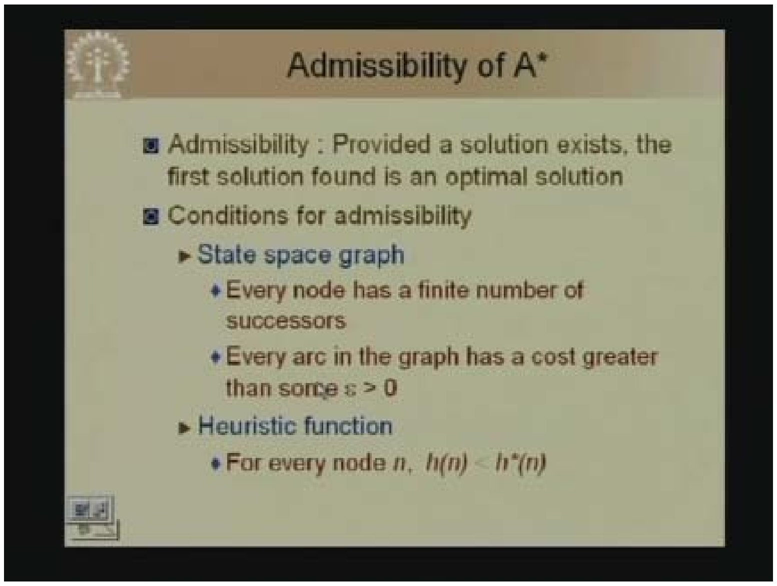


A* search : analysis

- Optimal – optimally efficient
- Complete
- Number of nodes searched still exponential in the worst case

We will show that A star is an optimum algorithm. That is it is also optimally efficient, it gives the optimum solution and it is also the optimally efficient algorithm. A star is complete. However, the number of nodes searched is still exponential in the worst case unless the heuristic is extremely or logarithmically accurate.

(Refer Slide Time: 39:46)



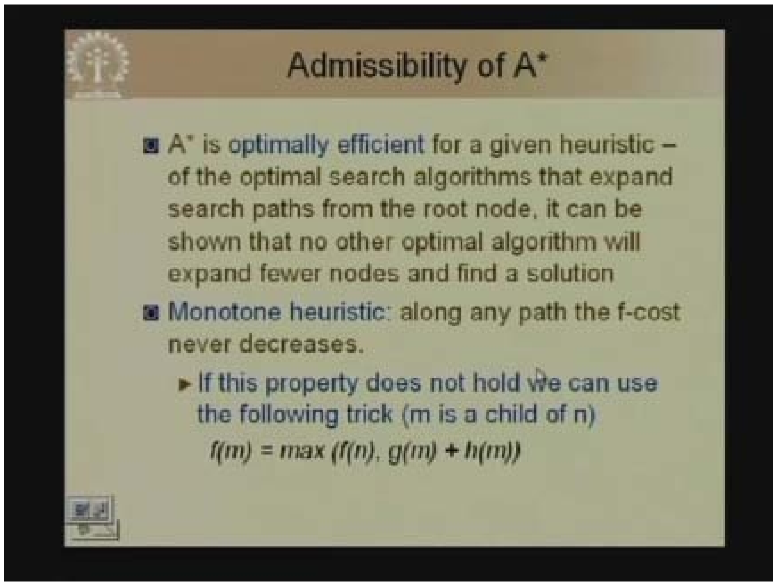
Admissibility of A*

- Admissibility : Provided a solution exists, the first solution found is an optimal solution
- Conditions for admissibility
  - State space graph
    - Every node has a finite number of successors
    - Every arc in the graph has a cost greater than some $\varepsilon > 0$
  - Heuristic function
    - For every node n, $h(n) < h^*(n)$

Now we will try to find out the condition on h(n) for which A star gives the optimum solution. So the property of admissibility is that, provided a solution exists the first solution found by the algorithm is an optimum solution. If A star can guarantee this we will say that A star is an admissible algorithm. So in order to show that A star is admissible we will try to find out the conditions under which A star is admissible.

Firstly, the state space graph should have this characteristic. Every node must have a finite number of successors. In the search tree or search graph every node must have a finite number of successors. Every arc must have bounded cost. That is, there exists an epsilon such that every arc cost is greater than epsilon. And thirdly we have a heuristic function h(n) which is always an underestimate of the actual optimum cost from n to goal. We denote that by h star n. And h star n is the cheapest cost of a path from n to a goal node. And our heuristic function h(n) must be an underestimate to h star n.

(Refer Slide Time: 41:26)



Admissibility of A*

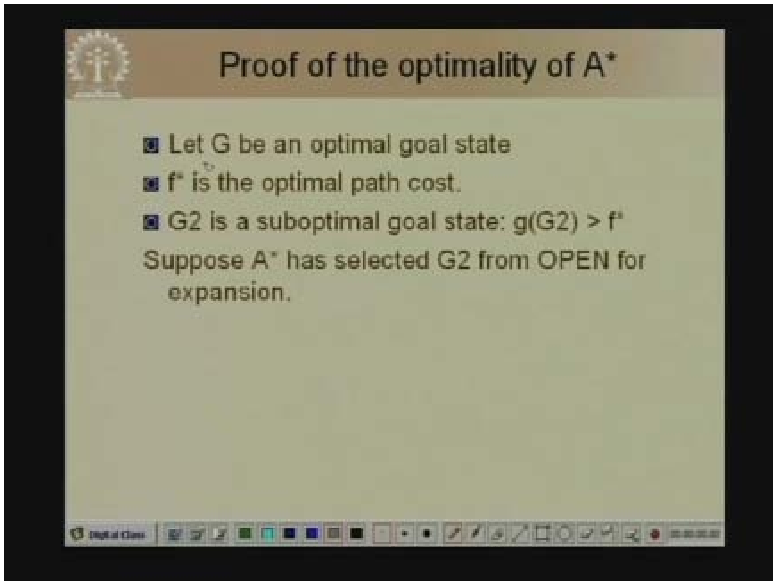
- A* is optimally efficient for a given heuristic – of the optimal search algorithms that expand search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution
- Monotone heuristic: along any path the f-cost never decreases.
  - If this property does not hold we can use the following trick (m is a child of n)
    $f(m) = max(f(n), g(m) + h(m))$

Now it can be shown that A star is optimally efficient. What do you mean by optimally efficient?
For a given heuristic, given a particular heuristic function h of any optimal search algorithm that you could have that expand search path from the root node it can be shown that no other optimal algorithm will expand fewer nodes than A star on the average and still always guarantee in finding a solution. Hence A star is an optimally efficient algorithm.

Secondly, we will consider a heuristic function which has the monotonicity property. That is, along any path the f cost never decreases. The f cost of different nodes along a path can only increase but it cannot decrease. So many heuristic functions h(n) satisfy this monotonicity property. However, even if the heuristic function does not satisfy this property we can easily enforce this property if we have an underestimating heuristic function by doing the following trick.

Suppose we have a node n and m is a child of n. So f(m) is normally g(m) plus h(m) and at node n we have f(n). Now we can say that we will use as f(m) the maximum value of f(n) and g(m) plus h(m). That is, if g(m) plus h(m) is smaller than f(n) we will use f(m) is equal to f(n) to ensure that the monotonicity condition holds. We can do this because the f value of a node is an underestimate of the cost from star t to goal through this node. The f value of this node m is an underestimate of the cost from star t to goal through n and m. So, if f(n) is an underestimate of this cost we can also use f(m) here without sacrificing the admissibility condition. Now let us look at the proof that A star is admissible.

(Refer Slide Time: 44:40)



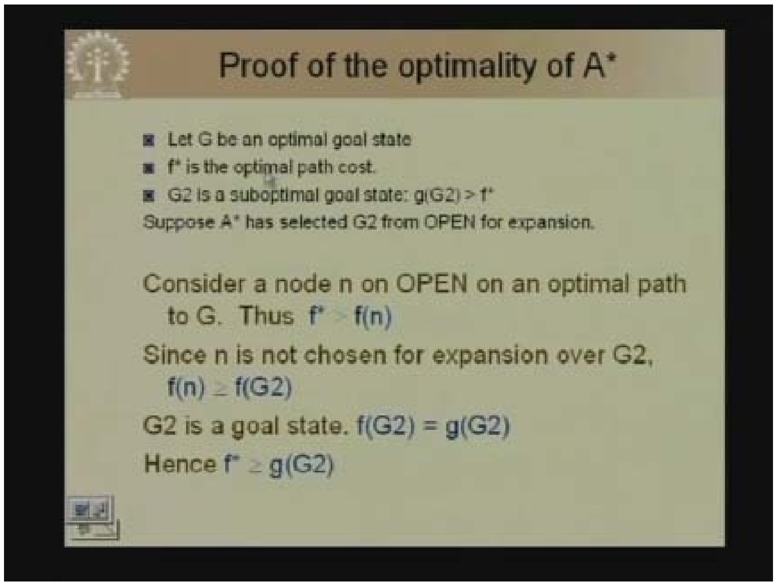
Proof of the optimality of A*

- Let G be an optimal goal state
- f^* is the optimal path cost.
- G2 is a suboptimal goal state: $g(G2) > f^*$

Suppose A* has selected G2 from OPEN for expansion.

A star always finds an optimum path to the goal. Suppose g is a goal state and f star is the optimal path cost for the algorithm A star so we have this star t state s, g is the nearest goal for which there is a path whose cost is f star. And suppose we have this other goal G2 which is sub optimum and there is a path from s to G2 and the cost of this path is greater than f star. Now we will try to show that it is not possible that A star algorithm will find G2 first before it finds g. In order to prove this let us star t with the opposite contradiction that, suppose A star has selected G2 from open for expansion.

(Refer Slide Time: 45:51)



Proof of the optimality of A*

- Let G be an optimal goal state
- f^* is the optimal path cost.
- G2 is a suboptimal goal state: $g(G2) > f^*$

Suppose A* has selected G2 from OPEN for expansion.

Consider a node n on OPEN on an optimal path to G. Thus $f^* \geq f(n)$

Since n is not chosen for expansion over G2, $f(n) \geq f(G2)$

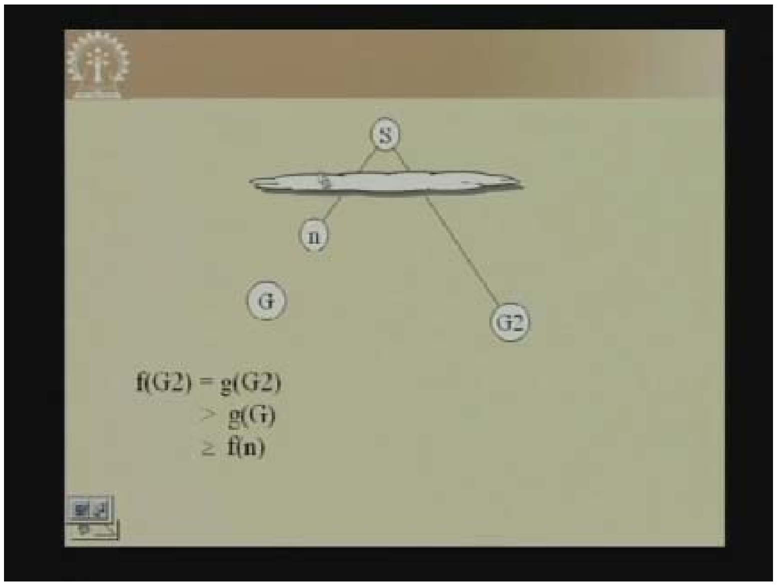
G2 is a goal state. $f(G2) = g(G2)$

Hence $f^* \geq g(G2)$

Suppose A star has selected G2 from open for expansion, now we have our node, this is my star t state, this is g and this is G2. Suppose if g is not on open there must be at least another node on this optimum path which is on open. So the node n of the optimum path must be on open. Now, suppose our algorithm A star has selected G2 for expansion and not selected n. If n is on the optimum path from s to g then f(n) must be less than equal to f star. That is, f star is the cost of the optimum path from s to g and f(n) which is an underestimate must be less than equal to f star.

Now, if G2 is chosen for expansion and n is not chosen for expansion it must be the case that f(G2) is less than equal to f(n). That is, f(n) is greater than equal to f(G2). Now, because G2 is a goal state f(G2) is simply g(G2) because h value of a goal state is 0. Therefore it follows from these three conditions that g(G2) is equal to f(G2) less than equal to f(n) less than equal to f star that is g(G2) is less than equal to f star. So, if A star has to select a goal node for expansion while the actual optimum goal has not yet been selected it must be the case that g(G2) is less than equal to f star. But this is a contradiction because we just said that G2 is a sub optimal goal so this cannot happen. That is, when A star selects a node for expansion and if that node happens to be a goal it must an optimum goal state.

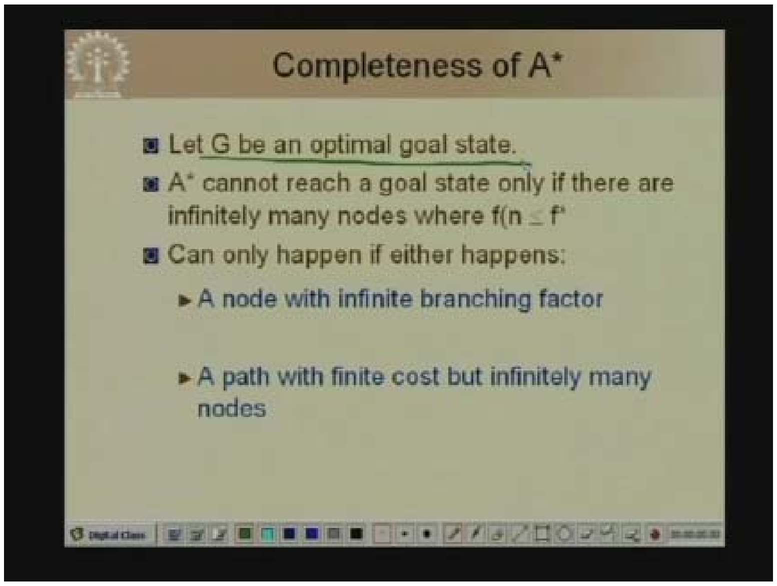
(Refer Slide Time: 48:28)



$$f(G2) = g(G2) > g(G) \geq f(n)$$

This is a sketch of the drawing. This is our s, this is g and this is the node on this optimum path which is on open when G2 is selected. And we just showed that f(G2) is equal to g(G2) which is greater than g (G) by resumption which has to be greater or equal to f(n) so this cannot happen. Therefore A star does find the optimum solution.

(Refer Slide Time: 49:16)

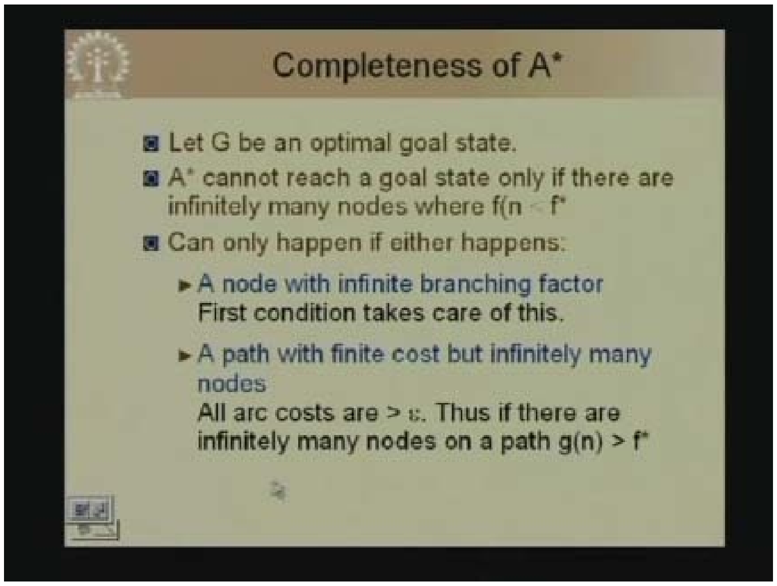


Completeness of A*

- Let G be an optimal goal state.
- A* cannot reach a goal state only if there are infinitely many nodes where f(n ≤ f*
- Can only happen if either happens:
  - A node with infinite branching factor
  - A path with finite cost but infinitely many nodes

Now we have seen that A star is complete is Optimum. We have to show that A star is complete. Now suppose g is an optimum goal state what are the conditions under which A star is not complete?
A star will not be able to read g only if there are infinitely many nodes for which f(n) is less than equal to f star. For expansion A star only selects those nodes which have f value is less than equal to f star. So A star will not be able to get to the goal node if there are infinitely many such nodes. This can only happen if either we have a node with infinite branching factor or we have a path with finite cost but infinitely many nodes. The first condition a node with infinite branching factor cannot happen because we assumed that the branching factor is finite.

(Refer Slide Time: 50:16)

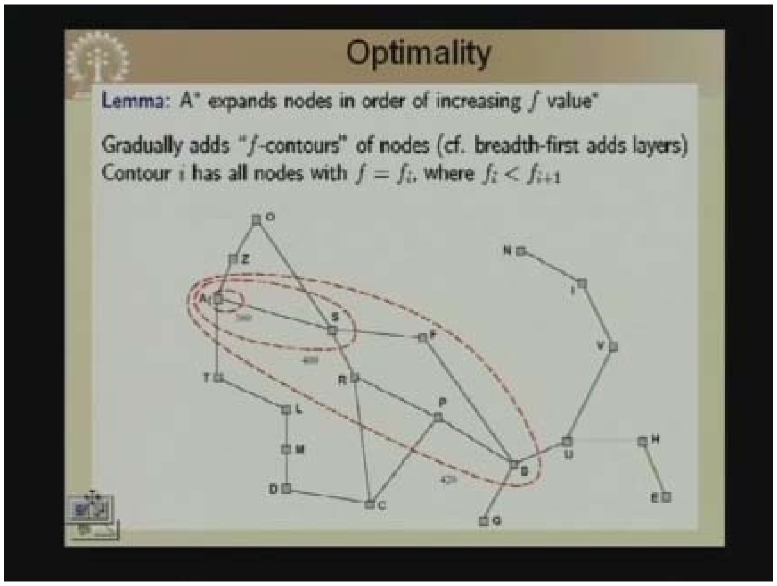


Completeness of A*

- Let G be an optimal goal state.
- A* cannot reach a goal state only if there are infinitely many nodes where f(n < f*
- Can only happen if either happens:
  - A node with infinite branching factor
    First condition takes care of this.
  - A path with finite cost but infinitely many nodes
    All arc costs are > ε. Thus if there are infinitely many nodes on a path g(n) > f*

The second condition cannot happen because we assumed that all our costs are bounded and they are always greater than epsilon. So there cannot be infinitely many nodes on the path. Thus A star is complete.

(Refer Slide Time: 50:38)



Optimality

Lemma: A* expands nodes in order of increasing f value*

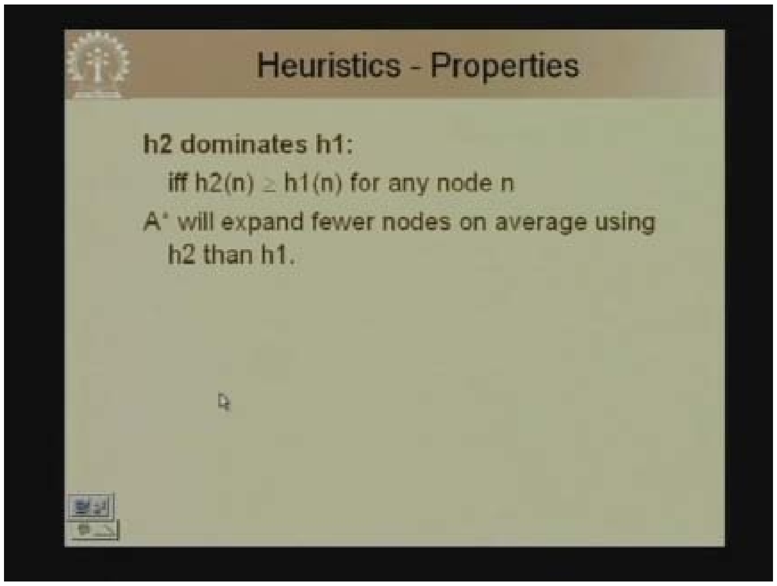
Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

This lemma states that A star expands only those nodes whose f value is less than equal to f star. In fact A star expands nodes in order of increasing f values. So we star t from this node this star t state and A star expands node with increasing value of f values.

Properties of heuristic functions:

Suppose we have two heuristic functions h2 and h1 we say that h2 dominates h1 if the value of h2 at any node n is grater than the value of h1 at that node n and we can show that A star will expand fewer nodes on average using h two than when using h1.

(Refer Slide Time: 51:40)



Heuristics - Properties

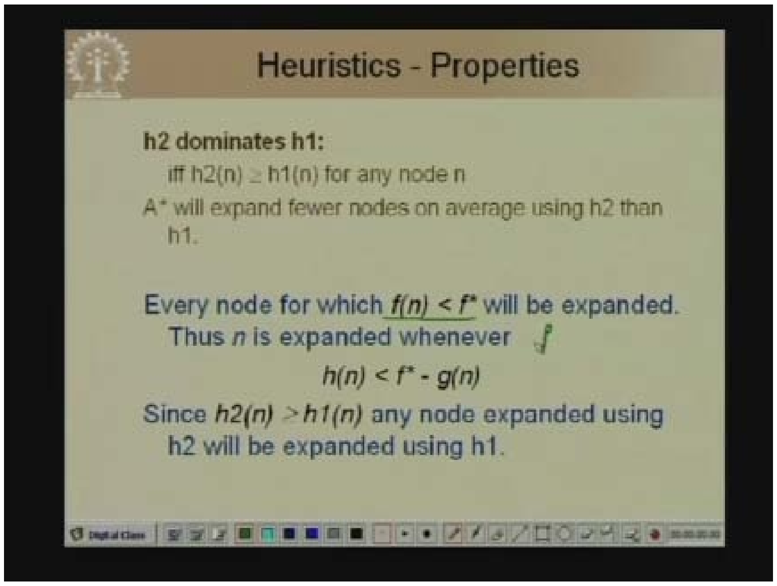
h2 dominates h1:

iff h2(n) $\geq$ h1(n) for any node n

A* will expand fewer nodes on average using h2 than h1.

Now let us show the proof sketch, the proof of this:
A star with h1 is the heuristic function expands every node for which f(n) less than f star. So every node for which f(n) less than f star will be expanded.

(Refer Slide Time: 52:03)



Heuristics - Properties

h2 dominates h1:

iff h2(n) $\geq$ h1(n) for any node n

A* will expand fewer nodes on average using h2 than h1.

Every node for which $\underline{f(n) < f^*}$ will be expanded.

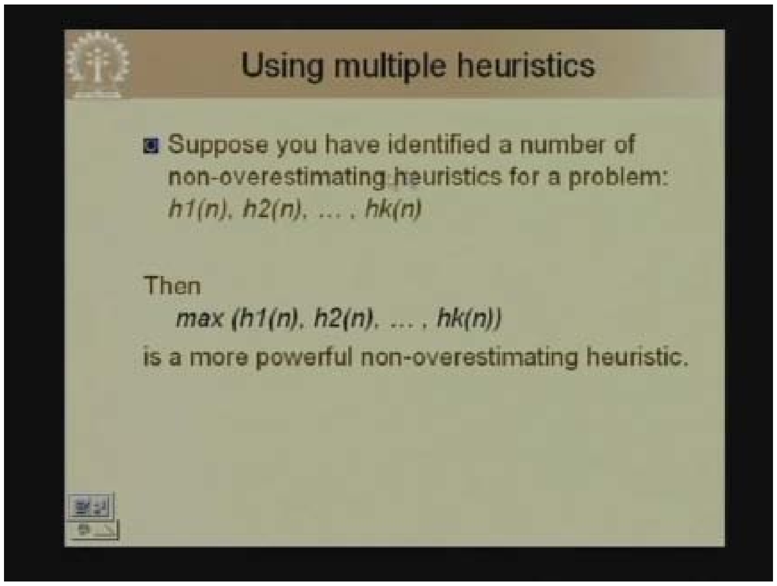
Thus n is expanded whenever

$$h(n) < f^* - g(n)$$

Since $h2(n) > h1(n)$ any node expanded using h2 will be expanded using h1.

Some nodes where f(n) is equal to f star some of these nodes will be expanded. A star with h2 as the heuristic will expand all nodes with f2 and f star. So, if h2(n) is greater than equal to h1(n) the nodes which A star with h2 will expand must be a subset of the nodes which A star with h1 can expand. However, depending on the execution those we might have some leeway for those nodes where f(n) is equal to f star. But on an average because A star expands those nodes whose h value is less than f star minus g(n). So, if h2 is greater than h1 at all nodes then A star with h2 will expand fewer nodes. So A star with h1 is preferred to A star with h2. So long as the heuristic is an underestimating heuristic we prefer that heuristic function which gives a higher estimate.

(Refer Slide Time: 53:30)



Using multiple heuristics

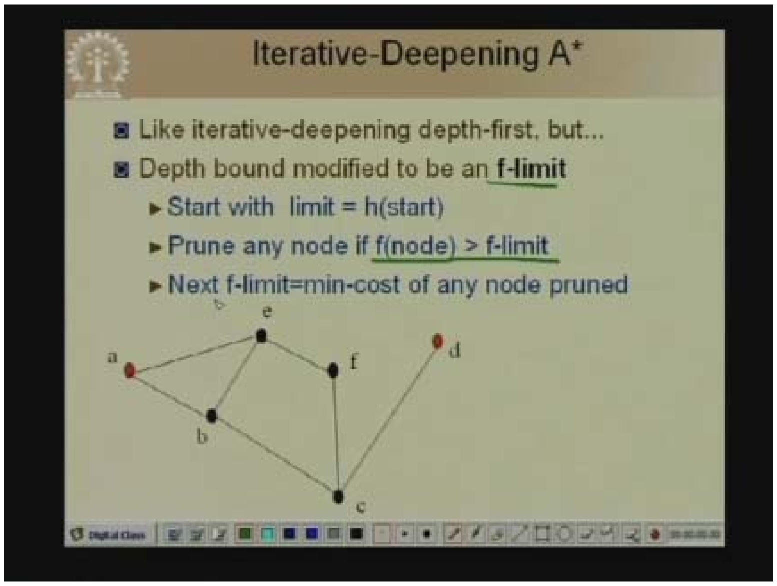
- Suppose you have identified a number of non-overestimating heuristics for a problem: *h1(n), h2(n), … , hk(n)*

Then
max (h1(n), h2(n), … , hk(n))
is a more powerful non-overestimating heuristic.

Suppose we have identified a number of underestimating heuristics for a problem where h1(n) h2(n) hk(n) and so on so we can combine this multiple heuristic functions by taking the maximum of h1 h2 and hk. So we can find the values of h1 h2 hk at that node and since all of them are underestimates we can take that value which is maximum. And this happens to be a more powerful non overestimating heuristics.

Now we will quickly look at a variation of A star which is similar like iterative deepening search. But instead of using a depth bound we use an f limit. Initially we star t with limit, the f limit is equal to h value of the star t node and then we do a depth first search. And we prune any node for which f value of the node is greater than the f limit.

(Refer Slide Time: 54:40)



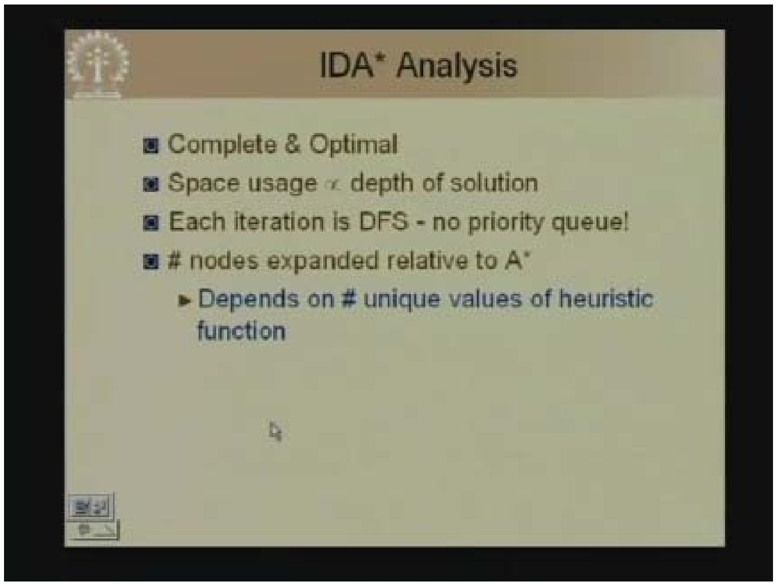
Iterative-Deepening A*

- Like iterative-deepening depth-first, but...
- Depth bound modified to be an f-limit
  - Start with limit = h(start)
  - Prune any node if f(node) > f-limit
  - Next f-limit=min-cost of any node pruned

We set the next f limit to be the minimum cost of any pruned node. So we star t from node a here we have to reach node d which is a goal node. We first set the f limit as 15 and expand this node in a depth first manner. Then we set the f limit as 21 and we expand these nodes and then we change the f limit and then we expand all the other nodes. So, iterative deepening A star can be shown to be complete and optimal.

However, because we use depth first search the space usage is proportional to the depth of the solution and it saves some space. The number of nodes expanded, it expands some nodes more than A star but usually it is of the same order as nodes expanded by A star. So in general if there are lots of possible f values in the search tree IDA star can generate square of the number of nodes that A star generates. But where many several nodes share the same f value IDA star will typically expand a constant order of nodes more than A star.

(Refer Slide Time: 56:04)

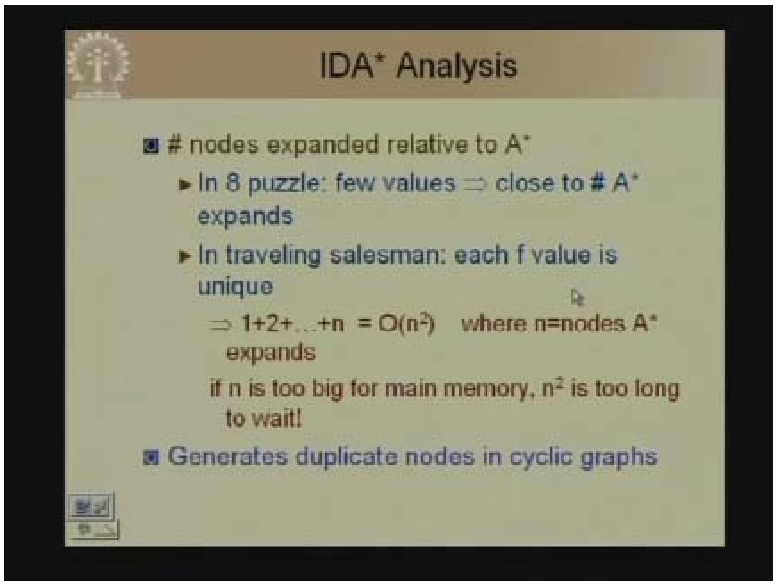


IDA* Analysis

- Complete & Optimal
- Space usage $\propto$ depth of solution
- Each iteration is DFS - no priority queue!
- # nodes expanded relative to A*
  - Depends on # unique values of heuristic function

Therefore in 8 puzzle we have few values of f because they are all integers, all the f values are integers so there are a few values so IDA star is quite efficient.

(Refer Slide Time: 56:30)

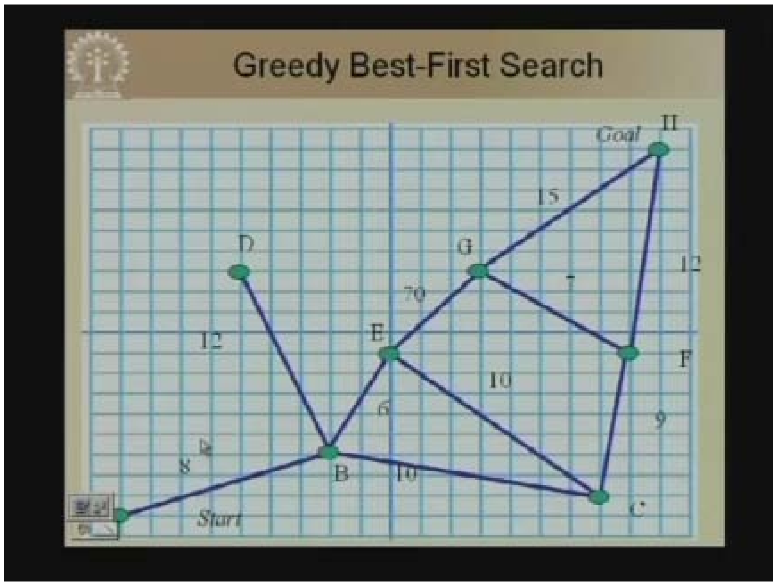


IDA* Analysis

- # nodes expanded relative to A*
  - In 8 puzzle: few values $\Rightarrow$ close to # A* expands
  - In traveling salesman: each f value is unique
    - $\Rightarrow 1+2+\ldots+n = O(n^2)$ where n=nodes A* expands
    - if n is too big for main memory, n^2 is too long to wait!
- Generates duplicate nodes in cyclic graphs

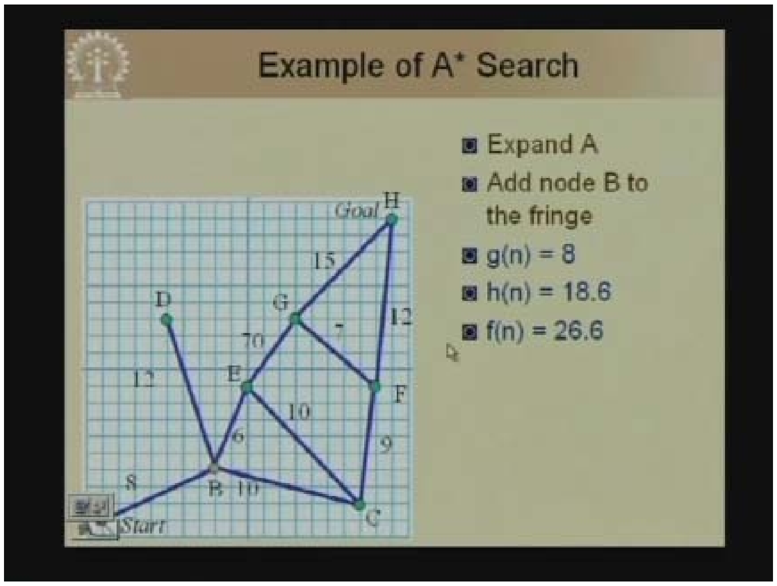
In traveling salesman problem each f value is unique if the path costs are real numbers. And the number of nodes expanded will be order of n square because it would be the case that at every iteration one more node is expanded. So the total number of nodes expanded is 1 plus 2 plus 3 plus up to n which is of the order of n square. And it is very difficult, if we are using depth first search we really cannot detect nodes which have been expanded before. Therefore it is not a very good choice if we have a search graph. Otherwise depth first search is very attractive because the space requirement is small.

(Refer Slide Time: 57:22)


Greedy Best-First Search

We will stop this lecture by quickly running greedy search as well as A star on this graph. We star t from star t state from a. If we do greedy search we will try to go to that node for which h value is smallest. Suppose we choose as h value the straight line distance from a node to the goal node. This is my star t node, this is the goal node in greedy search from a we will go to b, from b we will go to e, from e we will do g, from g we will go to the goal node h. and we get this path whose cost is 70 plus 15 plus 6 plus 8 is equal to 85, 91, 99.. This path is not optimal. Now let us take the same graph and run A star. We first expand node a and then we add node b to the fringe whose g is 8, h is 18.6 and f is 26.6.

(Refer Slide Time: 58:15)

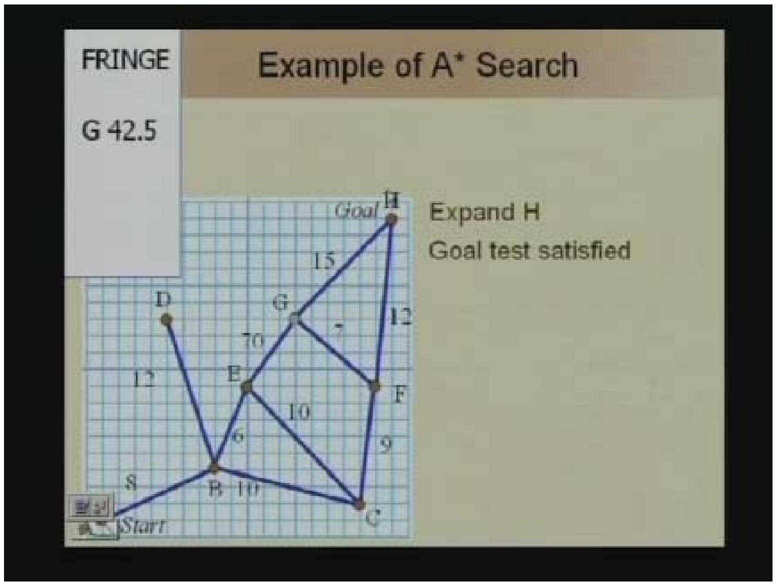


Example of A* Search

- Expand A
- Add node B to the fringe
- g(n) = 8
- h(n) = 18.6
- f(n) = 26.6

Then we expand B and then we add C, D and E to the fringe. For c the f value is 35.1 we put it in the fringe, we evaluate D, f value is 35.2. Then we evaluate e with f value 27.5 then we expand e add successors g and c evaluate g and g(n) is equal to 84, h(n) is equal to 8.5, f(n) is equal to 92.5. We add it to the fringe then we can take c whose f(n) is equal to 41.3 which is more than the f value of c which you already have on the fringe we discarded. And then we expand c add node f whose f value is 37. We put it in the fringe, then we expand d which has no children then we expand f add nodes g and h to the fringe. We evaluate g whose f value is 42.5 we replace the f value. Then we evaluate h whose f value is 39 we put it in the fringe we expand h which is a goal node and we have found a goal whose cost is 39 which is better than the cost of 99 which we found in the greedy search and this is the optimum path to this goal node.

(Refer Slide Time: 59:50)



FRINGE

G 42.5

Example of A* Search

Expand H

Goal test satisfied