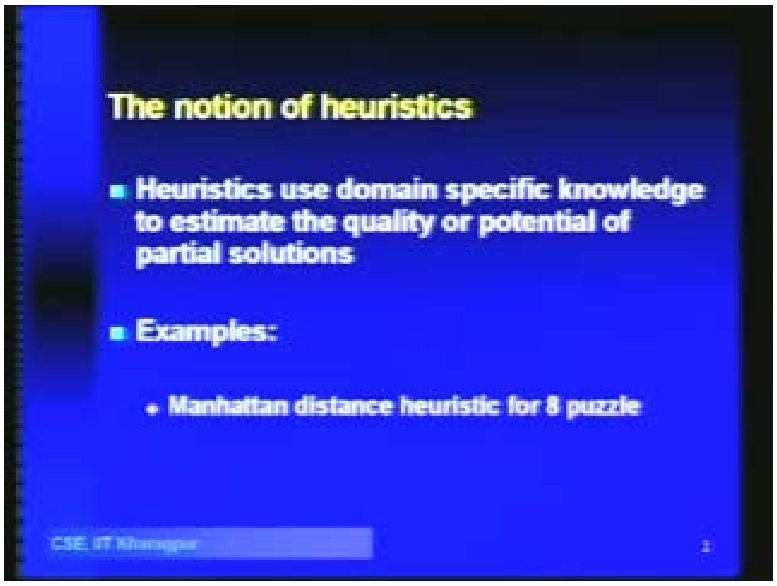
Artificial Intelligence

Prof. P. Dasgupta
Dept. of Computer Science & Engineering
I.I.T Kharagpur

Informed State Space Search
Lecture – 4

Okay, so we will start the section on informed state space search. In informed state space search- slides please. So, what we have here is- when we talk about state space search, we talk about the search space which is in the form of a set of states and set of state transition operators. Now, when we have informed state space search, it means that we have additional information, indicating the proximity of the goal from each state. So, as just outlined in the previous lecture, the notion of heuristics is as follows: that you have heuristics that use domain specific information to estimate the quality or potential of partial solutions, so that you know that if I have if the current state is the partial solution. Then, I know what is the potential of growing this into a full solution. What is the additional cost that I will require for doing that? Let us start with a few examples. One of most common heuristics for the 8 puzzle is the Manhattan distance heuristics.

(Refer Slide Time: 02:08)



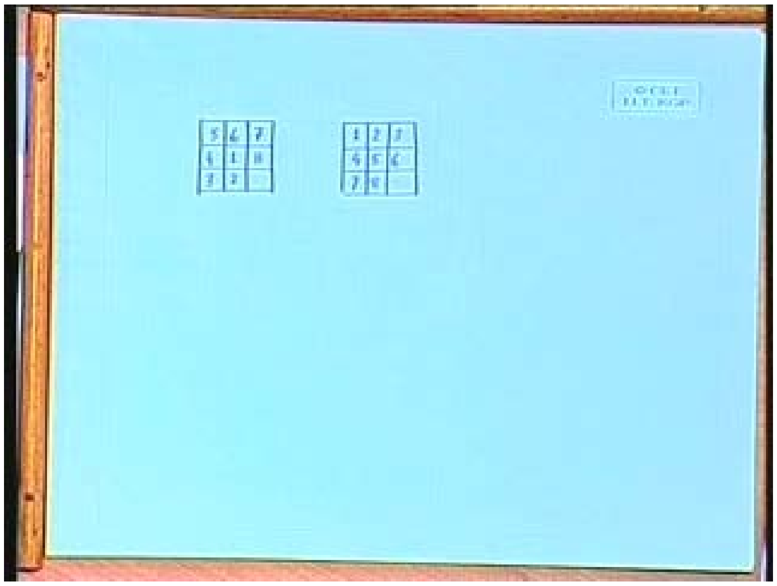
The notion of heuristics

- Heuristics use domain specific knowledge to estimate the quality or potential of partial solutions
- Examples:
  - Manhattan distance heuristic for 8 puzzle

CSE, IIT Kharagpur 2

Now, you remember the 8 puzzle, where you have all the tiles- the 8 tiles- arranged in a 3 by 3 square and we have to slide the tiles to bring it to some configuration. To see the Manhattan mode heuristics for this problem, let us say that the tiles are like this- that I have- and I want to reach the configuration- my goal configuration- which is known, again. Now, the heuristic- the Manhattan mode heuristic- says, that find for every tile, the Manhattan distance from its current position to its final position. And Manhattan distance is a commonly used term, which says that it is the distance computed in terms of the distance on the x axis plus the distance on the y axis. It comes from the fact that Manhattan apparently has all roads which are either east to west or north to south.

(Refer Slide Time 03:18)

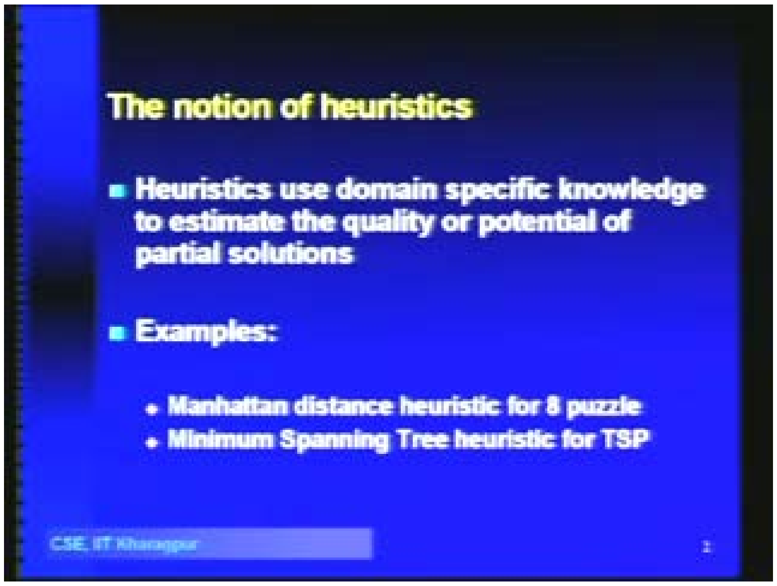


So, to travel from place c to place b, the total distance that you have to travel is your distance northwards plus your distance eastwards or westwards, right? For example, for this 5- the initial position is here and the final position is here. So, I have to take at least 2 slides to bring it here. Now, you see, that in order to bring this configuration to this, this tile 5 will have to be sided at least twice. It has to be slided at least twice. Similarly, the 6 the tile 6 is also at a Manhattan distance of 2. If you look at the tile 3, it is at a Manhattan distance of 1, 2, 3, 4, right? If you have to move this tile 3 from here to its final position, then no matter in what way you do it, it is guaranteed, that at least 4 slides will have to be made.

So now, if I just add up the Manhattan distances of each of these tiles together, the Manhattan distance that each tile has to take; if I add them up, then, can I say that I have lower bound on the number of moves that I have to make, in order to solve the puzzle? Yes. Why? Because every move, as we had said- what are the operators? Move the blank up, move the blank left, move the blank right, or move the blank down; but as a consequence of moving the blank, it is 1 tile which is being slided. So, every operation is sliding a single tile and we have seen that the Manhattan distance reflects the minimum number of slides that we have to make for a given tile, right? And at a single step, we are moving only 1 time, right?

If we have to move all these tiles that many number of times, then the total Manhattan distance is giving me a lower bound on the number of tiles that I have-, number of times that I have to move a tile, right? It could be actually much more; actual number of slides can be more, but I have lower bound on the number of slides that I have, right? (Student speaking). Yes, but when you are moving 1 tile, the others are not moving, right? When you are sliding 1 tile, you are having a feeling of pipelining the whole thing, but it is not actually pipelining. You have to do it 1 by 1. Every move that you do, has a unique cost, right? So, even if they all shift by one- the 4 tiles shift by one- it is a cost of 4, not one. That is why what we have is a lower bound. Does it clarify your query? Let us look at another heuristic. This is the- slides please, slides please.

(Refer Slide Time: 07:51)



The notion of heuristics

- Heuristics use domain specific knowledge to estimate the quality or potential of partial solutions
- Examples:
  - Manhattan distance heuristic for 8 puzzle
  - Minimum Spanning Tree heuristic for TSP

CSE, IIT Kharagpur 2

We have the minimum spanning tree heuristic for TSP. Now, this is what we do. We have the t an instance of TSP and we want to find out a lower bound on the cost of the tool. So, what we have is, we have these cities, okay, and we have to find out a tour to all these cities. So, what we are going to do is, we will first create a minimum spanning tree of this. And let us say that this is the minimum spanning tree. Let us say that the cost of this minimum spanning tree is c of s. c of the cost of the spanning tree, right, and let us say that the optimal cost of the tool that we are looking for is C*. Then, my first claim is that cs is less than C*. Now, let us convince ourselves that the cost of the minimum spanning tree is less than the cost of the optimal tour, assuming that all costs are non-0. If 0 is also allowed, then instead of less than, I will make it less than equal to.

Now, the reason that this is the case is, assume that you are given the minimum cost tour. If you take out any of those edges, you will have a spanning tree, right? Now, if the tour cost is less than the cost of the minimum spanning tree, then by removing 1 edge from the tool, you will further decrease the cost and have a spanning tree which has less cost than the minimum cost spanning tree, which is a contradiction, right? Therefore, we have cs less than C*. Is that all right? Okay. Let me repeat this. If I look at the minimum cost tool, suppose this is the minimum cost tool. If you remove an edge from the minimum cost tour, what are you left with? You are left with a spanning tree, right? Remember that in traveling salesperson problem, we are not allowed to visit the same city more than once, right?

Therefore, if you just chalk out the examining part of the tour after taking out any 1 of the edges, what you have is a spanning tree, right? Now, if the whole tour cost you C*, then the cost of this tree will be less than or equal to C*, right? And the minimum cost spanning tree can have cost only lesser than this; lesser than or equal to this. So, the minimum cost- spanning tree cost- is going to be less than or equal to C*, okay? Now then, my claim is that this C* is again less than twice of c s. Why so, why so? Because let us see the following thing: if I look at twice of cs, that means that I am traversing each of edges twice, right? Now, I will show you that I can create a tool which cost less than this. How? Yes, because this is- if I have to traverse like this, let us suppose I go from this city to this city, right? Then from this city to this city, right? Then, from here to here, then from here to here. Then, instead of going back here and then taking this, I will take the shortcut, right? And then, I will take this edge. Then, I will take this edge and then, instead of backtracking along these, I will go directly to the next city, right?

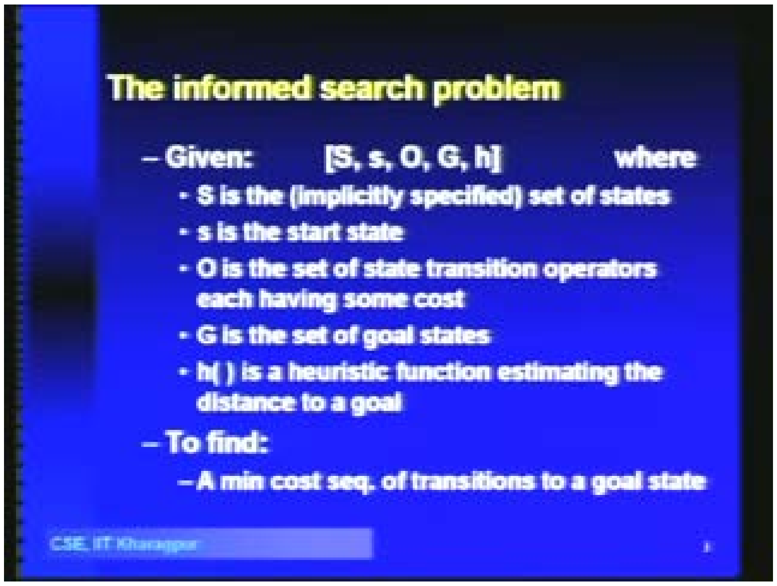
So, that could be this point, right, and then from here, again back here. Now, if these cities are in the Euclidian space, then you have the triangle inequality. That means that if I had to traverse this thing backwards, along this along all this thing, then the total cost that I would have to incur is going to be more than if I take this shortcut directly, from here to here. So, triangle inequality- the cost of all these edges; the sum of these edges is always more than the cost of taking the direct distance, from this city to this city. Therefore, this tour that I have constructed by just jumping from leaf to leaf like this, has a cost that has- that is less than twice of cs, right, and the optimal tour can only be better than this- better than or as maybe just as good as this, right?

Therefore, I have C* is less than equal to 2cs, right? Now, this gives us a nice thing- that if I take that if I compute the tour in this fashion, by computing the spanning tree and then constructing a tool like this, right, and then, I divide that by half- divide the tour cost by half, then I get a lower bound on the cost of the optimal tour. Yes or no? Find this tour; this tour cost is less than twice cs. So, if you take half of that, it is going to be less than cs, right? And so therefore, that is going to be a lower bound on the cost of the optimal tool, and then, you use that bound in your TSP, right?

Heuristics are fundamental to chess programs. Now, just to give you an idea about how chess programs will work, is that they will start from 1 board configuration. And if you would look at it in the naïve way, then you will start exploring all sequences of moves. And since the branching factor of chess is pretty high, so, you cannot really explore very deep. If you start exploring right down to the winning-losing configurations, that is going to take an enormous amount of time.

So, what chess playing programs try to do, is that they explore up to a few moves, look ahead. Say, 20 moves look ahead. They see all board configurations at a look ahead of 20 moves, right, and they evaluate each of those board configuration based on some heuristics. And then based on that heuristics, they decide which of the sequences of moves they will try to follow. There is more in it than I am saying here, and we will study little bit of that; about how you explore those kinds of trees when we come to game trees, but heuristics are very important there as well.

(Refer Slide Time: 17:34)



The informed search problem

- Given: [S, s, O, G, h] where
  - S is the (implicitly specified) set of states
  - s is the start state
  - O is the set of state transition operators each having some cost
  - G is the set of goal states
  - h( ) is a heuristic function estimating the distance to a goal
- To find:
  - A min cost seq. of transitions to a goal state

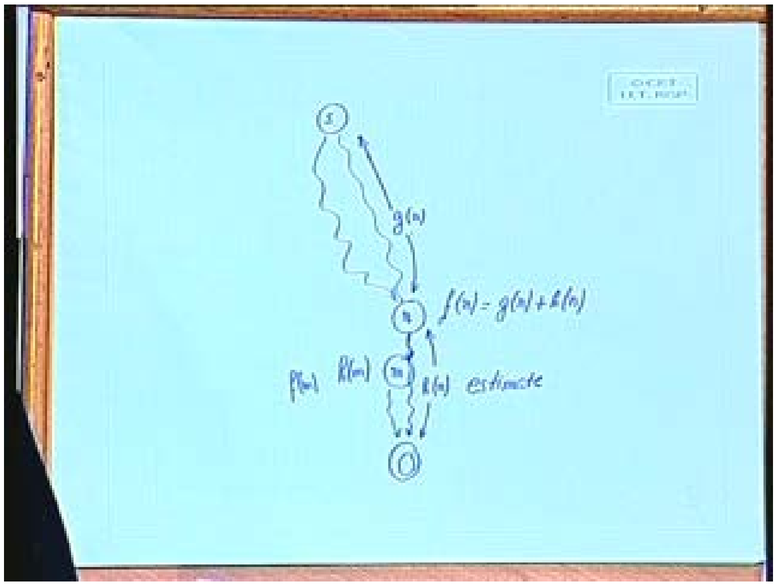
CSE, IIT Kharagpur

So, the informed search problem is like this: again we have our familiar 4 tuples, the state space, the start state, the set of transition operators, the set of goal states, and now, we have an additional function h, which is a heuristic function that estimates the distance to the goal. And our objective is as before: to find a minimum cost sequence of transitions to a goal state, right? So, the first algorithm that we will study here is A*. This is a very well known algorithm, which was proposed long back, perhaps around the time when you were born. So, this algorithm is of theoretical importance mainly, by the way, because, as we shall see, that it tries to maintain open and closed explicitly. And in practice, it will work for very few in very few cases and people actually we will study how to save the memory and still do something similar to A*.

That is where all the engineering comes up. So here, we have- in the initialize step, we set open to s and closed to empty, and we have 2 functions. 1 is gs, 1 is hs, and the cost of the state s will be denoted by fs. Let me explain what this gh business is. The g value of a state, at a given point of time, will indicate the minimum cost path from the start state to that state, right? We can draw it out on a picture. So, if we have a state here: this is the start state, and then I have found out a path to our state n, then this path cost is gn, right? hn is the estimated cost is the estimated cost. This is the estimated cost of traversing from n to a goal state- to any goal state- and I will maintain fn as the sum of gn plus hn. And what does that give us?

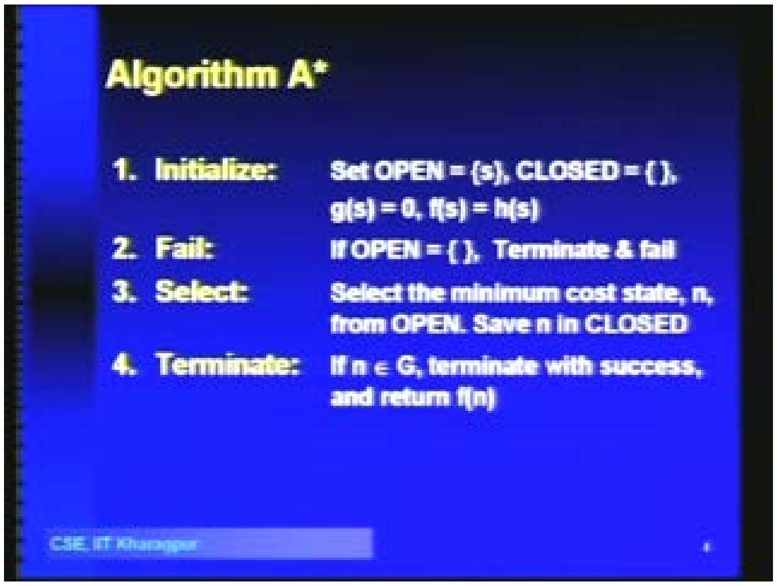
It gives us the cost of the best solution that goes through n, right? Now, you have studied “ “ No? in algorithms no? Okay. Please read it up. Please read up “ “ . Then, you will see that- then we can discuss that- what is the similarity with this? Anyway, we have gn here and hn here. Now, what are the things that can happen during the search? During the search, my g value can change, because I might find some alternative better path to the same state, right? And also, as you go down the path, when you from n to its successor m, the heuristic value of m might be better. So, it may give you a more accurate estimate of the distance of m to the goal. The cost can change the cost of the fm value- can change based on that- right? So, this is the basic notion of what we mean by the g and the h. Now, let us see how we use this in the algorithm. Initially, for the start state, the g value, gs is 0.

(Refer Slide Time: 22:15)



Because, from the start state to the start state, the cost is 0, and fs is the estimate of going from the start state to this state, because gs is 0. So, hs plus gs is the same as hs. Step 2: same as before, if open is- (Student speaking)- hs is the heuristic function which gives the estimate of the cost of reaching the goal from s, right? hs is the cost estimate of the cost of reaching the goal from s. Then, we have, the second step is: fail- if open is empty, then terminate and fail, just as we had before. Then again, we select the minimum cost state n from open, and save in closed, but here when I refer to cost, I am referring to fn. So, look at the f values of all the states in open, and pick up the 1 which has minimum f value. I will work out an example for this algorithm to make this thing clear. Then, step 4 is terminate. If n belongs to the goal set, terminate with success and return fn.

(Refer Slide Time: 23:57)



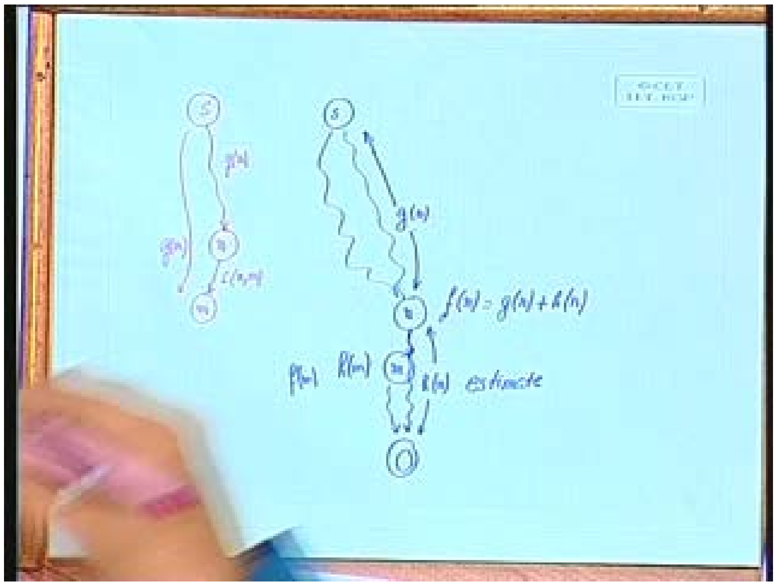
Algorithm A*

1. **Initialize:** Set OPEN = {s}, CLOSED = { }, g(s) = 0, f(s) = h(s)
2. **Fail:** If OPEN = { }, Terminate & fail
3. **Select:** Select the minimum cost state, n, from OPEN. Save n in CLOSED
4. **Terminate:** If n ∈ G, terminate with success, and return f(n)

CSE, IIT Kharagpur

Should I bring this back? Is it okay? All right. So, let us go to the expand step. For each successor m of n, if m does not belong to open or closed, which means that we are expanding it for the first time- this is the first time that we are visiting the node. Then, we set gm to be gn plus cnm. Now, why is that so? If you have you have this the From the start state s, I have gone to n and this was my gn. Now, when I generate a successor m, the g value is going to be this whole cost. So, it is gn plus cnm. cnm is the cost of applying the operator to move from n to m, right? So, gn plus cnm- that is the value of gm, right?

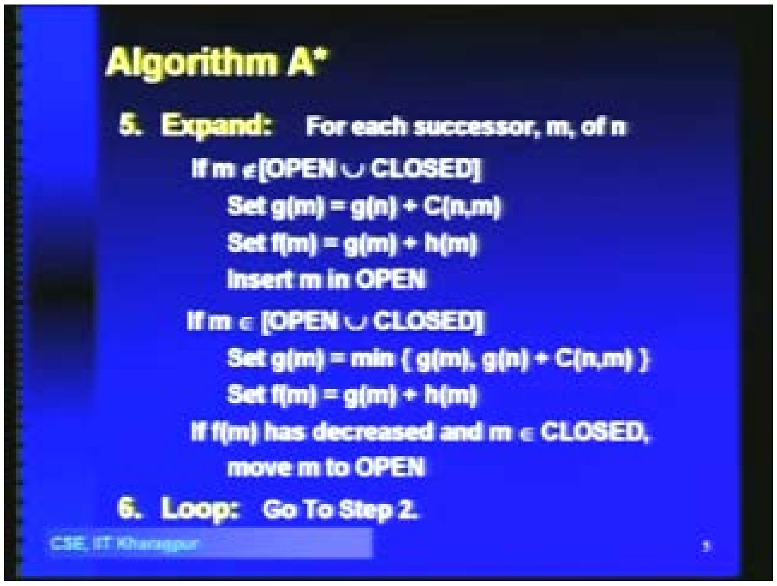
(Refer Slide Time: 25:18)



So, coming back to here, we have gm is equal to gn plus cnm, and then we set fm equal to gm plus hm, where hm is the estimated cost of reaching the goal from m, right? Then we insert m in open. Again, if m already belongs to open or closed, then, we set gm is equal to minimum of gm and gn plus cnm. We see whether we have obtained a better path to reach the state m, right? hm is going to remain the same. So, we find out what is the best gm that we have got so far, and then add that with hm, to get the new cost of fm. Now, if fm is decreased- which means that we have indeed found a better path to that state- and we find that m belongs to closed, then we will move m to open. Otherwise, we will simply update the cost of m.

If m is there in open, we will simply update its cost to the new value of fm, right? Now, this is similar to what we have done previously for the uniform cost search, except that now, states can move from closed to open, because of the heuristic function. The heuristic estimate, at an earlier stage, could have been worse than, as you progress further along some paths, and may have been good always, along some other paths. So, therefore, something which was considered not so good at some point of time, suddenly might become good, because the other paths which were promising, as we went down, their heuristic costs increased, and we found that no, these are not good enough, right?
We will see examples of this nature of nodes coming back from close to open, okay? Then, let us go further down. Otherwise, we go to step 2, right? Fine.

(Refer Slide Time: 27:51)



Algorithm A*

```
5. Expand:   For each successor, m, of n
      If m ∉ [OPEN ∪ CLOSED]
         Set g(m) = g(n) + C(n,m)
         Set f(m) = g(m) + h(m)
         Insert m in OPEN
      If m ∈ [OPEN ∪ CLOSED]
         Set g(m) = min { g(m), g(n) + C(n,m) }
         Set f(m) = g(m) + h(m)
      If f(m) has decreased and m ∈ CLOSED,
         move m to OPEN
6. Loop:   Go To Step 2.
```

CSE, IIT Kharagpur

Let us work out an example to see this. So, I have this graph, which is the same as the 1 that we had seen before, and let me quickly write down the costs that we had. This was the cost that we had: can you see clearly? Now, in addition to this, let us assume that we have some heuristic values that we find from the nodes. So, that is some function which tells us at each state- what is the estimate of the goal. So, let us say, in 1, I have the heuristic value of 12 and at 2, I have heuristic value of 10. That means that at when I am in state 2, I have an estimate that from 2 to goal will cost me at least 10. That estimate is given to us. That is the heuristic function that is given to us, right?

Now, note that though I am writing down these numbers besides the states, you do not know these values until you are actually- you have generated this state, and applied the heuristic function on that state. So, for example, if you have reached a particular state of 15 puzzle, then you can use the Manhattan heuristic on that state, to find out a lower bound on the number of moves that will take you to the goal. Similarly, in case b, after you have generated a partial tool, it is then that you know, that what is an estimate of the remaining tour. So, these are heuristic values. And what will be the heuristic value of the goal? 0.

Now, let us start with our lists. We will have open here and closed here. Initially, open will contain 1, with a cost of- no, no, no, no. 12, because you have the estimate.
So, your cost initially is not 0 from the start state. It is the fn value which is gn- gs plus hs, so gs is 0 always at the start states. hs is 12, which I have from here. So, I have this thing with a cost of 12, right? When the first step, I will expand the state 1 and put it in closed. So, I will have the state 1 with a cost of 12, which goes into closed and that will lead me to generate the successors 2 and 5. 2 will come with a cost of how much? 12. Why? Because the g value is this 2 and the h value is this 10, and the f value is g plus h. So, it is 10 plus 2, 12, right? What does this 12 indicate? It indicates an estimate of the cost of a solution path that goes through 2, right? This 12 does not tell me the solution from 2 is 12, no?

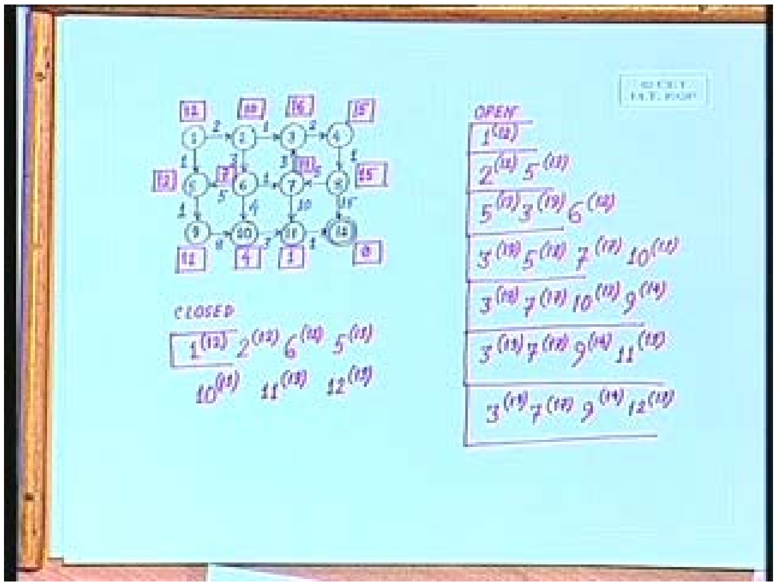
The estimate of the solution from 2 is 10, but I have already incurred a cost of 2 for coming from the start state to this. So, the total solution cost is 10 plus 2, 12. The estimate of the total solution cost is 12, clear? And then, here, I have how much? 13, right? So, in the next step, I will be picking up the one with minimum f value. That is 2. So, 2 with a cost of 12 comes out, and what do we have here? We will have 5 with a cost of 13. And for 2, we will now have 3 with a cost of 19. That is 16 plus the g value, which is 3. Now see, it is not that every time I will add these things up. What I am doing is, see, the g values of these nodes are being maintained. So, I know that the g value of 2 is 2. So I take- when I expand 2 to get 3, I will take the gn value, which is 2 added with this 1 to get 3. That is going to the g value of 3. g3 is 3, and h is what I compute when x takes 3.

So, g plus h is 19 and also, I will have 6 with a cost of 12. That is, 7 plus 5. g value is 5, h value is 7. Next, I will be picking up 6. So, 6 with a cost of 12 is here. I have 3 with a cost of 19, 5 with a cost of 13 and now I am expanding 6. So, will have 7 with a cost of 17. Yes? Why 17? Because the g value is 6 and the h value is 11. So, 17. And also, I will have 10 with a cost of 13, and also 5. But what will be the cost of 5? It is coming as 22. The current value of 5 is 13, so that is no use. We just discard it. We do not do anything about it, because the new cost of that- we have found the new g value is actually larger than the existing g value. So, this is no good, right?

Now, I have 2 nodes having cost of 13. Let us say without loss of generality, then I will pick up 5, okay? If I pick up 5 with a cost of 13, then what do I have here? I have 3 with a cost of 19, then 7 with a cost of 17, then 10 with a cost of 13. And then I have 9 coming up, by expanding 5, which comes with a cost of 12 plus 2, 14, right? Next step, I will be picking up 10 with a cost of 13. So, if I pick up 10 with a cost of 13, then I will have this 3 with a cost of 19, right? 7 with a cost of 17, 9 with a cost of 14 and by expanding 10, I will get 11 with a cost of 13, right?

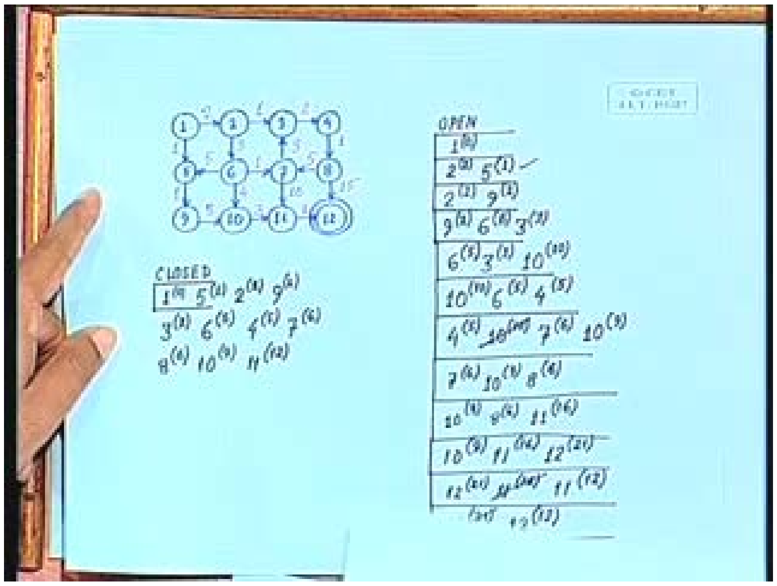
Next step, I will be picking up 11 with a cost of 13, right? That will give me 3 with a cost of 19, 7 with a cost of 17, 9 with a cost of 14 and 12 with a cost of 13. Next step, I will be picking up 12 with a cost of 13 and since that is the goal so we terminate, and we declare that the minimum cost is 13. Minimum- previously also, we had found that the minimum cost was 13.

(Refer Slide Time: 37:49)



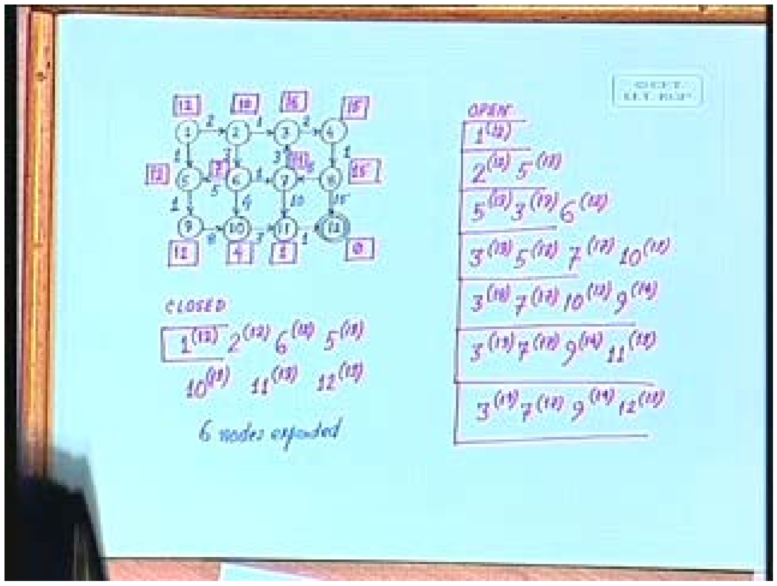
Now, let us compare this with what we had for uniform cost search. So, for when we worked out the one for uniform cost search, you can check that we had; this is the one that we had for uniform cost search.

(Refer Slide Time: 38:08)



Then, I had 11 nodes expanded, those are the nodes that you have not closed. You had 11 nodes expanded, and if you compare that with the heuristic search, when we had made use of this heuristic information, you see that we now have- how many?- 6 nodes expanded, and then when we picked up the seventh node, we found that it was a goal. So, we had to actually expand 6 nodes, right? It has reduced the total effort of the search, but how? How did it deduce it?

(Refer Slide Time: 38:59)



What it did here was that, there were some nodes which looked extremely promising at the beginning, like, if you look at 3, it came up here with a cost of only 3, but then, if you have to follow along 3, then if you go to 3 to 4, well, the cost is still good. It is 2 plus 1 plus 2. So, if you just ignore the heuristics, then this cost would be 5, which is still less than the cost of the optimal solution. So, that means 3 would have been expanded, 4 would have been expanded, right? 8 would have expanded, and if you recall, if you go back to the previous example, you will see that all these nodes were actually expanded, when we did not have the heuristic information. But the heuristic information here told us that look from 3, you will have a cost of 16, at least.

Therefore, at this point, only the cost of 3 became 19, and as you can see, that 3 never got expanded. 3 never got expanded, as a result, 4 never got generated, 8 never got generated, and they were also not expanded, right? So, what this essentially did was, it performed a look ahead, and was able to tell us, a priori, that this path is not going to be good. Now, let us do some analysis on the set of nodes that will be expanded by these algorithms. If you look at the uniform cost search algorithm, then I will say that all those states which can be reached with a cost less than the cost of the optimal solution; all those states will have to be expanded by any algorithm.

Now, let me reiterate. Suppose I you give me an algorithm a; you give me an algorithm a and you claim that this algorithm is always going to find the optimal solution. We do not have any heuristics. We are talking about the uniform cost search paradigm, so I do not have any heuristics. I am given a problem which means a start state and a set of states transition operators, and you give me an algorithm a and claim that this fellow is always going to give me the optimal solution. And then, I am trying to say that look, I think that the complexity of your algorithm will be such, that it will expand all states which have a cost less than the minimum solution.

You know why? Because suppose there is some state. So, this is our start state, and there is some state here- some state n- and the cost of n the cost of n is less than C*. And if your algorithm a does not expand n, then I am going to give the algorithm a another instance of the problem, where the entire state space will be similar, except that just below n, I will add a goal, right, and whose cost will be say cn or just cn plus some epsilon, where cn plus epsilon is also less than C*. I can always find such an epsilon. And then, because nothing else has changed in the state space, the algorithm a will be unable to find this goal, because it is not expanding n, and if does not expand n, then it will never discover this goal. Therefore, it will give you a sub-optimal solution. It will still give you C*, but you have a goal which has better cost.

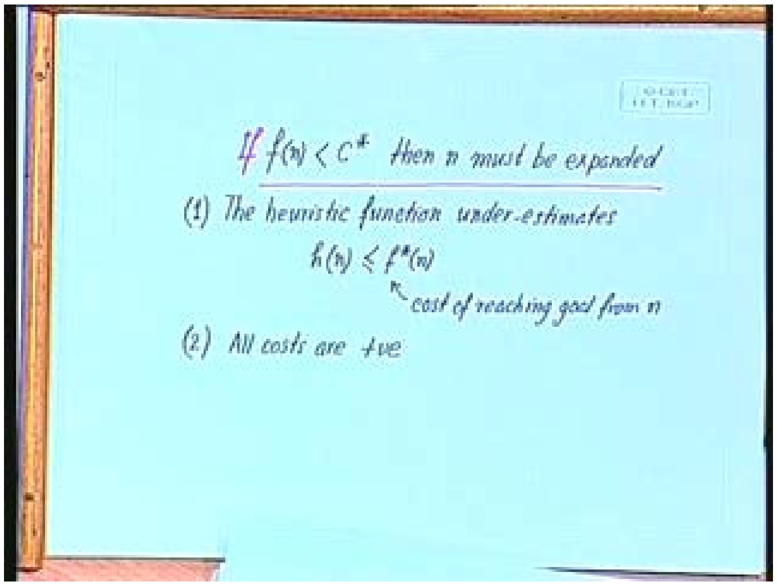
Now, is this analysis clear? Once again? Okay. My claim is that if cn is less than C*, and C* is- what?- optimal cost, then n must be expanded. This is my claim. Then, n must be expanded. This is my claim. Now, how do we establish this claim? We say that, let us assume that we have an algorithm a, which does not expand n. So, let algorithm a does not expand n, right? Then, what we can do is, we can keep the remaining state space identical. We do not make any change to the remaining state space, except that below n, we just add a goal and give it a cost which is between cn and C*. We can always do that, because cn is less than C*, so you can have some epsilon which you add to cn, and then this h cost is that epsilon, right?

So, this goal will have cost cn plus epsilon. Now, from the point of view of a, nothing has changed, because the entire remaining state space is similar, and in that in that scenario, a was not expanding n. So, a will still not expand n, and if it does not expand n, then it will not see this goal. It will not be able to see this goal, unless it expands n, clear? Understood? (Student speaking.) You create another state space, yes. You create another state space, which is similar to that previous state space, except that we have a goal just below this. (Student speaking). No, not yet. So, for this class of problems, where we do not have heuristic functions, the claim is that all states which have cost less than C*, will have to be expanded. And if you think of it, Dijkstra's does exactly that, right?

It always expands the minimum cost state in your frontier. Therefore, when you have when you have found the goal, then all the states that you have in your frontier or in your heap, they all have cost more than the cost of the __. And because you are dealing with positive edge cost, in case of Dijkstra's, so you know that by expanding the remaining set of states, you are not going to ever come to another state which has lesser cost than the ones that you already have, right? Now, so this is what we have in the case of uniform cost search.

What will happen in case of A*? In A*, we have also the heuristic function. The heuristic function gives us an estimate of the cost to the goal. Now, can I characterize the set of states which A* will expand, given a heuristic function h? (Student speaking). Yes, so let us make the claim first, then we will reason about it. The claim is that if I have fn less than C* then n must be expanded. See, again, at this point of time, we are assuming 2 things: we are assuming, one, that the heuristic function under-estimates, that is, hn is less than or equal to, where f star n is the … And second thing is, if you go by this rule, then what we have here is, if fn is less than C*, then n must be expanded by an algorithm, because if it does not- if it does not expand the state n- then, what we can do is, we can again create another state space, which is exactly similar to the existing state space. But below n, we will put another goal.

(Refer Slide Time: 51:04)



If $f(n) < C^*$ then n must be expanded

(1) The heuristic function under-estimates

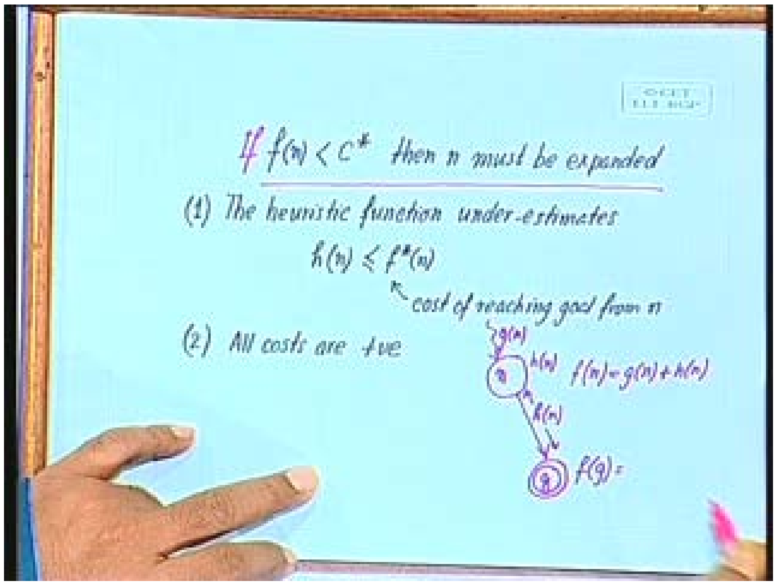
$h(n) \leq f^*(n)$

cost of reaching goal from n

(2) All costs are +ve

We will put a goal, and because fn is less than C*, so therefore, what we can do is that there will be some heuristic cost hn here, right? So, we will associate that heuristic cost with the cost of the transition to the goal. So, suppose we have what? The transformation that we are going to do is as follows: we have n here, and then, we will create a new goal, say g, right? Now, this had an hn component, and this had some gn component from above. So, fn was gn plus hn. What I am going to do is, I am going to make this h cost h n, equal to hn. In that case, what is going to be the f value of this? fg.

(Refer Slide Time: 52:06)



If $f(n) < C^*$ then n must be expanded

(1) The heuristic function under-estimates

$h(n) \leq f^*(n)$

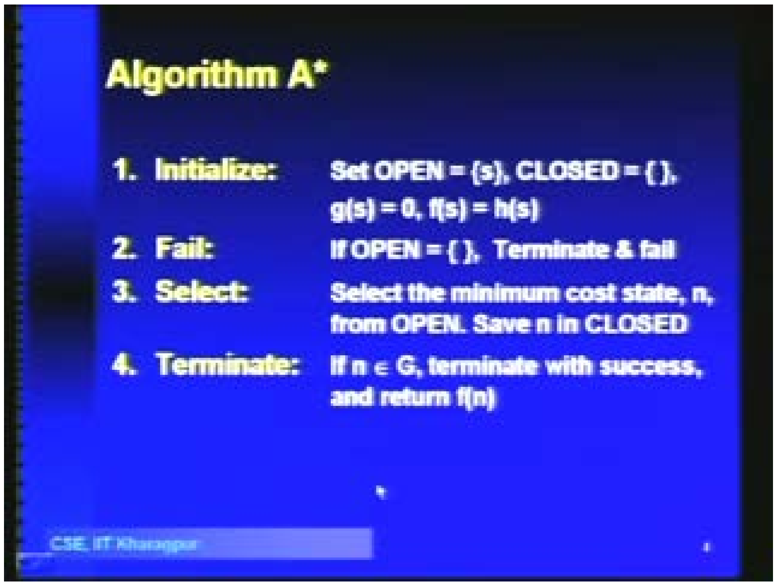
cost of reaching goal from n

(2) All costs are +ve

It is the- gn will be the g for this state, will be this gn plus hn. So, it is going to be fn and the h value is 0. f g will have the same value as f n. So, now I have another goal. I have a goal which has the same cost as the node n, and the algorithm, if it does not expand n, it will not be able to discover this goal, right? So, by a similar reasoning, we establish that if you have a cost- if you have a state whose cost is less than that of C*- then, every algorithm which guarantees finding the optimal solution, will have to expand that, right? Now, note that I have not written less than or equal to. I have written strictly less than; the ones which are strictly less are surely going to be expanded, but if you have less than or equal to, then we do not know, right?

Now, I have not mentioned here, if you come back to the slide algorithm A*, then, here I was selecting the minimum cost state n from open, right? Now, if you have many states having the same cost, which one will we select? What we do there is, if you have many costs, many states with the same cost, select the one which has minimum g value- among those states, which have the same f value, select the ones select the one which has minimum g value, because the others have already incurred a more cost in terms of g and we do not know that accuracy of the heuristics, right? Okay.

(Refer Slide Time: 54:20)



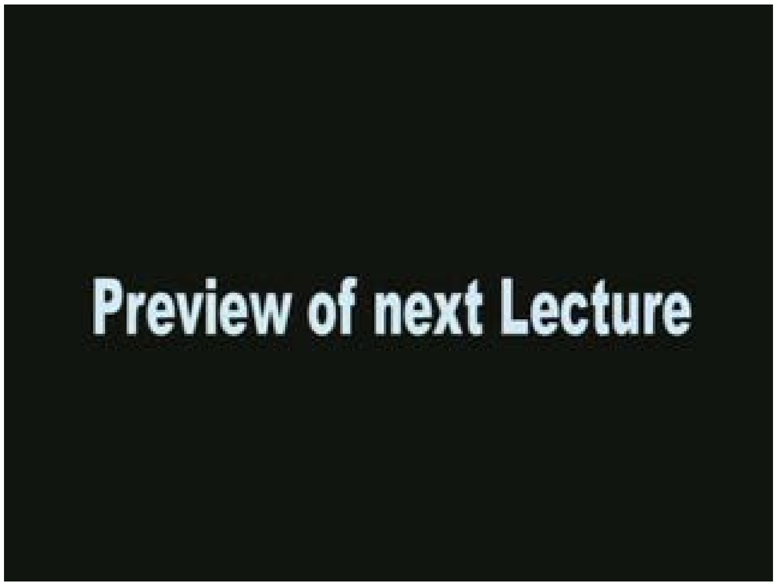
Algorithm A*

1. **Initialize:** Set OPEN = {s}, CLOSED = { }, g(s) = 0, f(s) = h(s)
2. **Fail:** If OPEN = { }, Terminate & fail
3. **Select:** Select the minimum cost state, n, from OPEN. Save n in CLOSED
4. **Terminate:** If $n \in G$, terminate with success, and return f(n)

CSE, IIT Kharagpur

So, with that, we will conclude this lecture. In the next lecture, I will start by analyzing some results of A*, and then we will study how we can create variants of A*, which will work better than A*. A* does not well work very well in practice. That is because it requires too much of memory. It is storing the whole of open and the whole of closed and it eases up too much of memory. So, it does not work in practice, but there are variants of that which are used. So, we will study some of those in the next class.

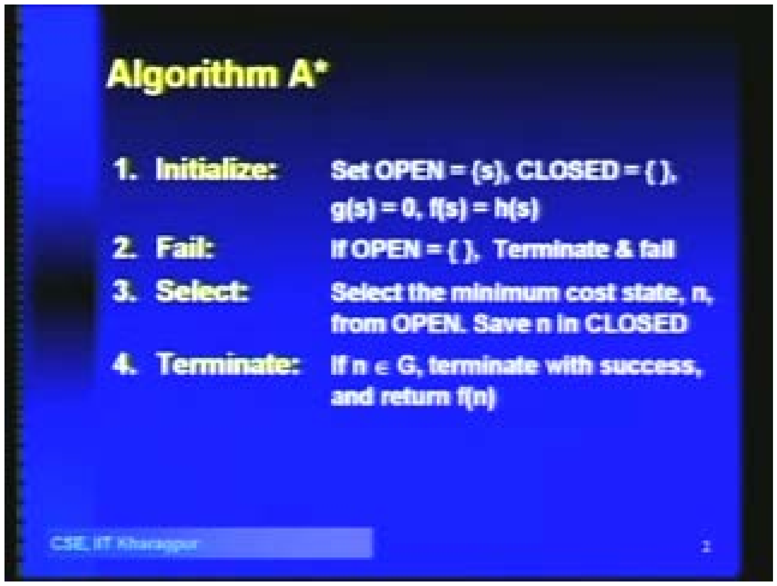
(Refer Slide Time: 55:01)



Preview of next Lecture

We will continue with our discussion on A* and heuristic search engine from this class onwards. We will this The topic of this lecture is heuristic search- A* and beyond. So quickly, to recap what we had done in the last class: we studied the algorithm A* which maintains 2 lists- open and closed- and also 2 functions. One is the g value, which computes the distance of the state from the start state, and the h value, which is the heuristic estimate of the distance of that state from the goal state. And fs is the sum of gs and hs, and that gives us the estimated cost of a solution, which goes through the node n. So, the first step was: if open is empty and we have still not yet found the goal, then we terminate with failure, otherwise we select the minimum cost state n from open and save it in closed. If the selected state is a goal state, then you terminate with success and return the f value of that state as the cost of the goal.

(Refer Slide Time: 56:35)



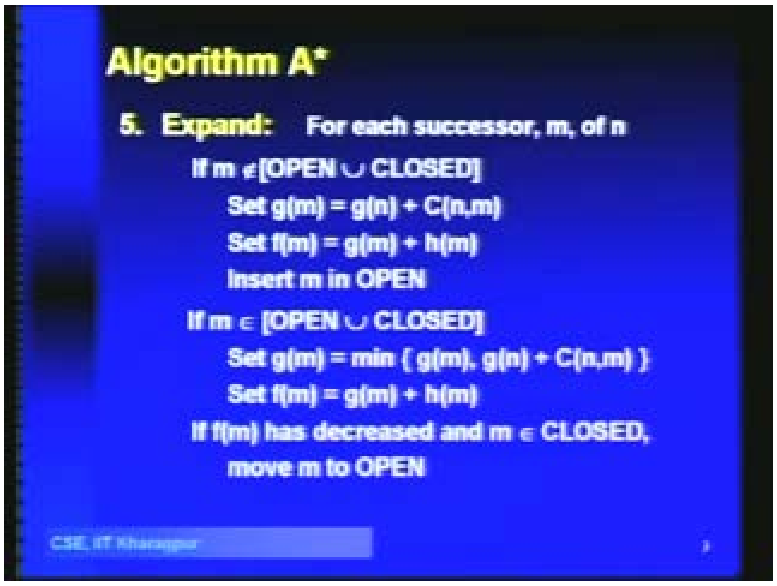
Algorithm A*

1. **Initialize:** Set OPEN = {s}, CLOSED = { }, $g(s) = 0$, $f(s) = h(s)$
2. **Fail:** If OPEN = { }, Terminate & fail
3. **Select:** Select the minimum cost state, n, from OPEN. Save n in CLOSED
4. **Terminate:** If $n \in G$, terminate with success, and return $f(n)$

CSE, IIT Kharagpur 2

Otherwise, we expand the node n, and to generate the set of successors and for each successor m, we compute its cost, based on the g value of that node and the h value of that node, and if the node already belongs to open and closed, we update it only if the cost is decreased. And if the node is already in closed and its cost has decreased, then you must bring it back to open. Now, in uniform cost search, we had seen that if you have only positive cost, then you cannot have a case where a node comes back from closed to open.

(Refer Slide Time: 57:14)



Algorithm A*

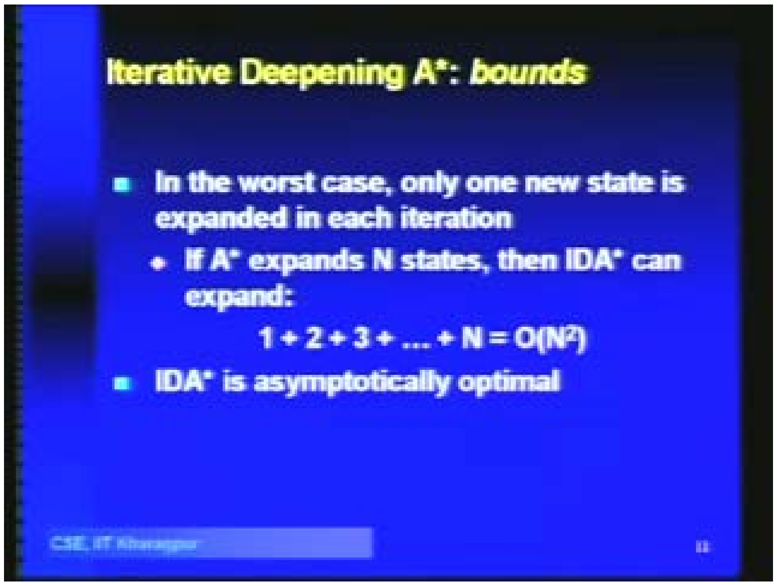
```
5. Expand: For each successor, m, of n
      If $m \notin$ [OPEN $\cup$ CLOSED]
         Set $g(m) = g(n) + C(n,m)$
         Set $f(m) = g(m) + h(m)$
         Insert m in OPEN
      If $m \in$ [OPEN $\cup$ CLOSED]
         Set $g(m) = \min \{ g(m), g(n) + C(n,m) \}$
         Set $f(m) = g(m) + h(m)$
      If $f(m)$ has decreased and $m \in$ CLOSED,
         move m to OPEN
```

CSE, IIT Kharagpur 3

Here, in the first iteration, you may expand one. Second iteration, you are expanding 2 states. Every iteration, only 1 new state is getting expanded. Third iteration: 3 states are getting expanded and so on, until you expand all n states. And that is the set of states which A* will expand. (Student speaking). Because you are not saving them, because we are not saving them. So, we are again; we are doing a DFDB on the state space, with the new cost cutter, right? So, it is all in the interest of saving space, because space is the thing which will kill you, in this kind of state spaces.

So, this gives you in the worst case, as you can see, order of n square, where n is the set of states which A* expands. n is the set of states that I was mentioning- all states with cost less than C*. This is going to be, in the worst case, quadratic in time, as compared to A*. The time increase is only quadratic, but the space is exponentially saved, because it can grow in order of b to the power of m, where b is the branching factor, but here you are doing in linear space, right? So, it is asymptotically optimal.

(Refer Slide Time: 58:40)



Iterative Deepening A*: *bounds*

- In the worst case, only one new state is expanded in each iteration
  - If A* expands N states, then IDA* can expand:
    $$1 + 2 + 3 + \ldots + N = O(N^2)$$
- IDA* is asymptotically optimal

CSE, IIT Kharagpur

Okay. So, there are several extensions of this basic memory bounded search strategies. Maybe sometime later, in some later lectures, I will just touch upon the other kinds of strategies that we have. But from the next lecture onwards, we are going to move into problem reduction search and game trees.