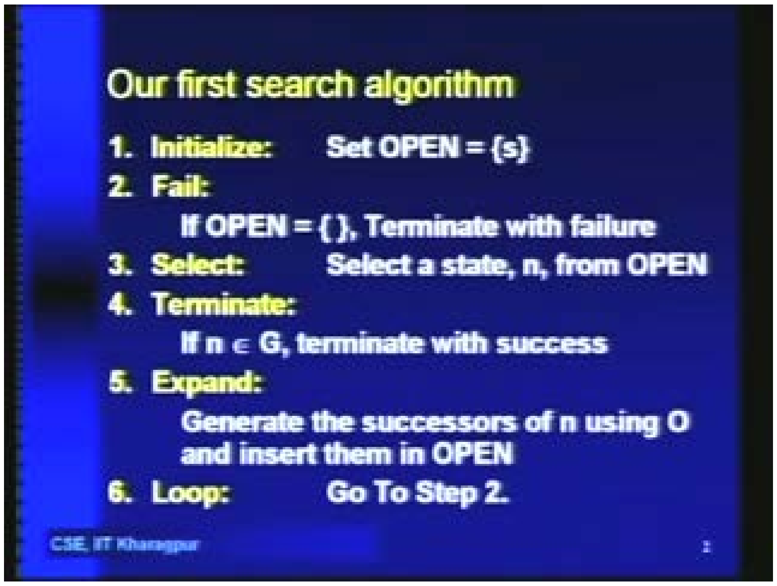
Artificial Intelligence

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

Searching with Costs
Lecture – 3

Today, we will start with the next chapter- on searching with costs. In the last class, what we had done was- we studied the first search algorithm, which was as follows: that we start with initialize, and we were maintaining a list called open. We put the start node in s. Then, if we find at some point of time, that open is empty and we have still not found the goal, then we terminate with failure. Then, select a state n from open, terminate if n is a goal; then, we terminate with success. Otherwise, we generate the successors of n using O and insert them in open, and finally, go to step 2. So, what we are essentially doing is, we are progressively maintaining a frontier of the nodes. And in each iteration, we are picking up a node from the frontier, expanding that. So, that extends the frontier and we continue in this way.

(Refer Slide Time: 01:58)



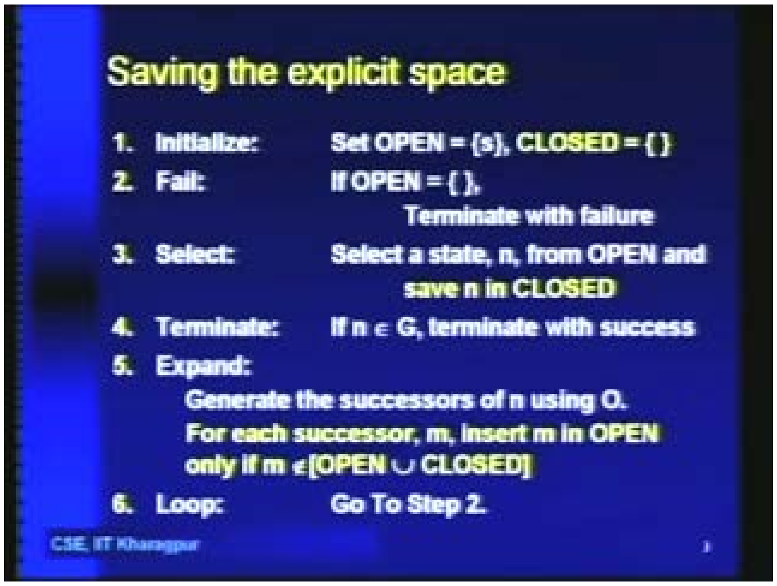
Our first search algorithm

```
1. Initialize:   Set OPEN = {s}
2. Fail:
      If OPEN = { }, Terminate with failure
3. Select:       Select a state, n, from OPEN
4. Terminate:
      If n ∈ G, terminate with success
5. Expand:
      Generate the successors of n using O
      and insert them in OPEN
6. Loop:         Go To Step 2.
```

CSE, IIT Kharagpur 2

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

(Refer Slide Time: 02:42)



Saving the explicit space

```
1. Initialize:   Set OPEN = {s}, CLOSED = { }
2. Fail:         If OPEN = { },
                     Terminate with failure
3. Select:       Select a state, n, from OPEN and
                     save n in CLOSED
4. Terminate:    If n ∈ G, terminate with success
5. Expand:
     Generate the successors of n using O.
     For each successor, m, insert m in OPEN
     only if m ∉ [OPEN ∪ CLOSED]
6. Loop:         Go To Step 2.
```

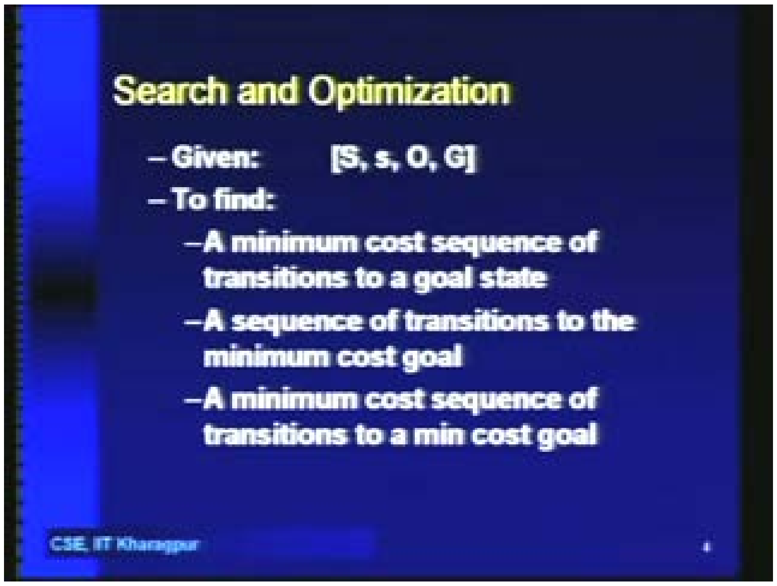
CSE, IIT Kharagpur

See, this just shows the difference with the previous algorithm: the only changes that I have made are highlighted in yellow. So, we are now maintaining another list called closed. When we select a state n from open here, we save it in we save the state in closed, right, and there is one difference here; that when we generate the successors of n using O, we will check whether those successors already belong to open or closed. If it already belongs to open or closed, then we will not insert it in open again, otherwise, we will insert it in open, right? Okay. Now, can you tell me what changes I need to make in this algorithm to also ensure, that after I find the goal, I can trace the path from the start state to the goal? Any additional things that I need to maintain?

Now, this algorithm is clear right? Yes or no? So, in this algorithm, I just want to add a few more features, so that I am able to maintain the path from the start state to the goal. (Student speaking.) What we can do is, with every node, with every state, we will maintain a parent pointer, and when will we set the parent pointer? Yes, when- (Student speaking.) not exactly. Whenever we are expanding the node Whenever we are expanding a node n and generating a successor m, we will make the parent pointer of m equal to n, right? So, whenever we are pushing a node into open, we will maintain- for every node that is there in open- we will maintain the parent pointer, right?

Later on, we will see that this is a useful thing to do, because we will be looking for optimal paths, at the minimum cost paths to a node. There you can arrive at a node from 2 different paths. So, when you come along one path and then later on you find a better path, you will shift the parent pointer to the new parent. We will go into that as we progress further. What we have done so far is, we have studied just the basic search algorithm, where we have a list called open, a list called closed and we are transferring nodes from open to close as we are expanding there, right? So far, we have not talked anything about costs. Now, let us see- how do we introduce costs into the picture?

(Refer Slide Time: 05:59)



Search and Optimization

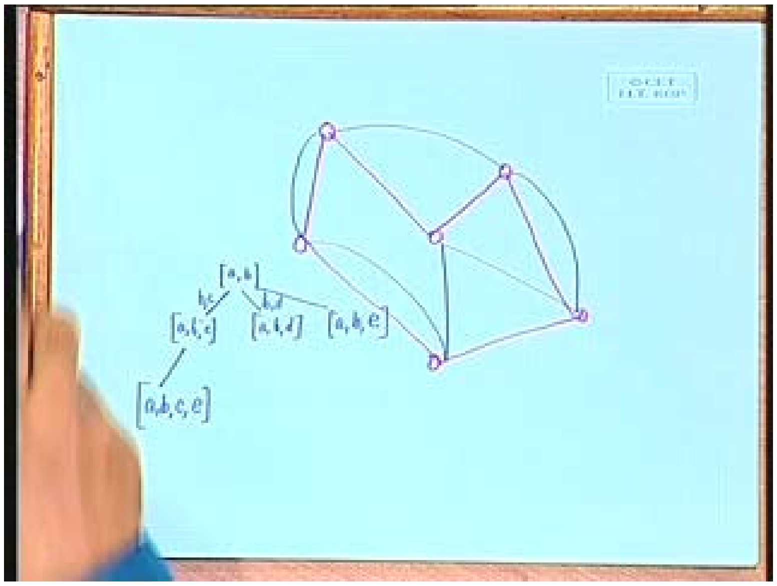
- Given: [S, s, O, G]
- To find:
  - A minimum cost sequence of transitions to a goal state
  - A sequence of transitions to the minimum cost goal
  - A minimum cost sequence of transitions to a min cost goal

CSE, IIT Kharagpur

So, when we have to search and optimize, or, in other words, we have to find out the shortest path or the minimum set of operators to reach the goal, then, we will have additional issues here. Besides these basic 4 tuple, along with each operator, we will have some cost associated. If you just want to reduce the number of steps, then, the cost can be one for every operator. In other cases, an operator can have an associated cost. Like- for example, you know the traveling salesperson problem. In the traveling salesperson problem, we have a set of cities, and the job is to find a tour for the traveling salesperson, to cover all the cities and then come back to the starting city, right? And there can be various different tours.

For example, this is one tour. Another tour could be: I go from here to here, then here to here, and like this, like this and like this, like this, right? So, if you look at a formulation- a state space formulation of TSP, what should we do? We will start with one- what will be your state? A state could be the partial tour that we have already covered. So, I have already covered city AB and I have not yet covered anything else. From here, I can pick up city C or I could go from A to B, and then, from there to D, right? Or ABE, right? And then, from here again, I could go to ABC and then, say, E, right? So, this is the formulation. And what are the operator costs here? The operator cost is the cost of traversing from the from B to C here. The edge cost BC is the operator cost here, for here, the edge cost BD is an operator cost, right?

(Refer Slide Time: 08:32)

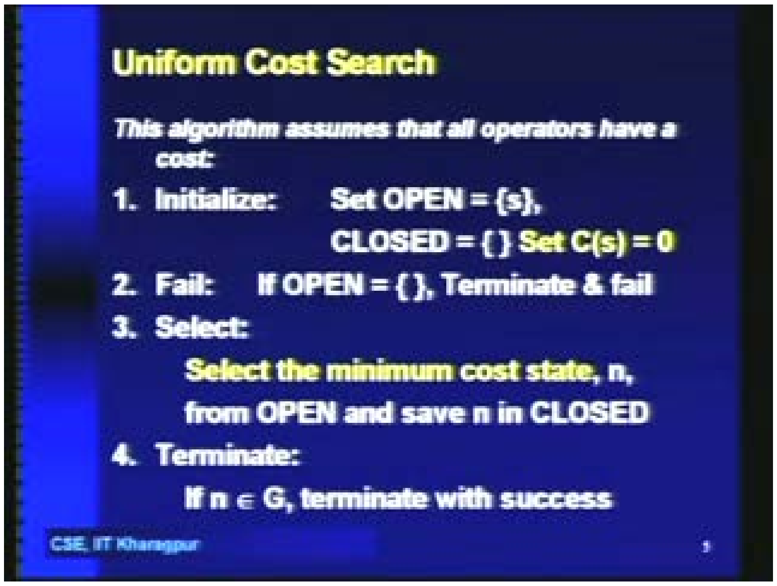


When we actually look at an optimization problem, then, the each step of the optimization is going to come with some cost, right? And that is what we will augment our basic formulation unit. So, what we have is this 4 tuple, and we have to find- there can be several things that we may want to find. One is, we might ask for the minimum cost sequence of transitions to a goal state- any goal state- and we just want a minimum cost sequence of transitions to that, right? In this case, for TSP, the goal state will be defined as a valid tool. And then, if we define the cost operators like so, then, the minimum cost sequence of transitions to the goal state will also give us the minimum cost goal, right?

We could also ask for a sequence of transitions to the minimum cost goal. What would that mean? Okay. Suppose, we define the goal of the 8 queens problem, as any arrangement of the 8 queens. So, every state is a goal, right? And if there are 2 queens attacking each other, then, we associate a cost with it. Essentially, what we are saying is- place the queens in the board in such a way that the minimum cost is incurred, and the minimum cost incurred can be 0 when all the queens- none of the queens are attacking each other, right? There, we will not associate any cost with the operators, but we will associate the cost with the goal.

We will see some kinds of search algorithms where we start- where all states are goals- and we will iteratively improve the goal, until we reach the minimum cost goal. Another option is a minimum cost sequence of transitions to a minimum cost goal. So, operators have some cost, goal has also some cost and we want to find the minimum cost sequence of transitions to the best goal. Now, these problems are somewhat related, between each other- the same problem can be formulated in one or more nodes. Now, let us see how we augment our existing algorithm- the one that we have studied so far- to take into account this cost. You will see that it is very familiar to an algorithm that you already know. Let us start.

(Refer Slide Time: 11:13)



Uniform Cost Search

This algorithm assumes that all operators have a cost:

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

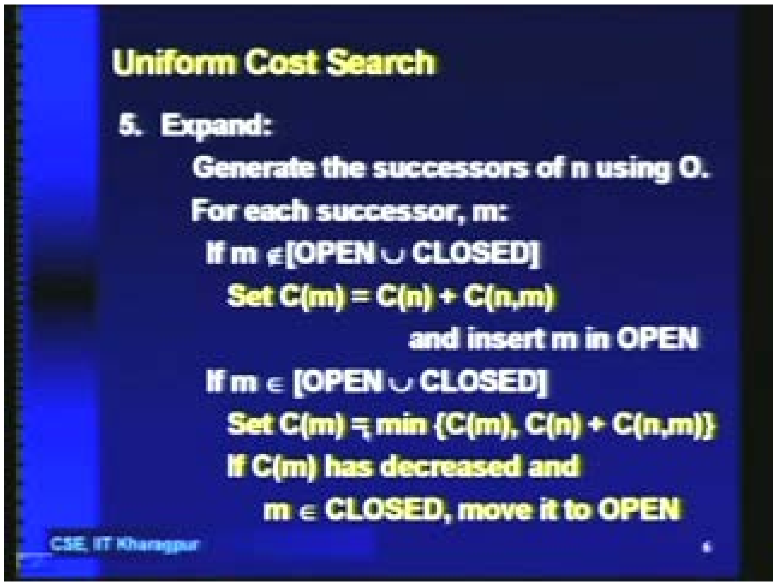
CSE, IIT Kharagpur 5

What we are doing here is: like previously, we are having open, which contains the start state s. We have closed, and we set the cost of every state will be denoted by C of that state. So, c is the cost function for the state. Initially, the cost of the start state is assigned to zero. Then, again, like previously, if open is empty, we terminate and fail. Otherwise- now here is the difference. We will select the minimum cost state n from open. Out of all the states in open, we will pick up the one which is minimum cost state. Later on, we will talk about what we do in case of tie breaking. We will come to that later. Assume now, that all costs are distinct. So, we just pick up the minimum cost state. We pick it up from open and save it in closed.

Now, in order to do this, what will be the data structure- one possible data structure, for open? A heap, right? But we will see that we will need more than a heap, because the costs of the states may change. So, in a heap, if the cost of some states inside the heap changes, then you will have to rearrange the heap you have written. There are ways of doing that. The 4th step is the same as before- we terminate, if we have found the goal.
Now, see that we will pick up the minimum cost state n from open, and if that state is a goal, then only we terminate with success, right? Because we are always picking up the minimum cost node from open. So, this part is simple. Only the selection from open is based on the minimum cost.

Now, when we expand, this is what we do: we generate the successors of the state n and then, for each successor m, if it does not belong to open or closed, then we will insert it in open. But with what cost? We will insert it in open with the cost cn, that is, the parent's cost, plus the cost of the operator to take it from n to m. This gives us the cost that we incurred to come up to state n plus the additional incremental cost to go from n to m. And we will assign that cost as cm and insert m in open. Now, suppose n already belongs n already belongs to open or closed. It is already there in open or closed. Then, we will again we will set cm to the minimum of its original cost and the new cost that we compute by cn plus cnm. So, if we have arrived at the same state m, through another path, which has cost lesser than the original path, then we will replace the existing cost by this cost. Clear?

(Refer Slide Time: 14:40)



Uniform Cost Search

```
5. Expand:
      Generate the successors of n using O.
      For each successor, m:
        If m ∉ [OPEN ∪ CLOSED]
          Set C(m) = C(n) + C(n,m)
                    and insert m in OPEN
        If m ∈ [OPEN ∪ CLOSED]
          Set C(m) = min {C(m), C(n) + C(n,m)}
          If C(m) has decreased and
             m ∈ CLOSED, move it to OPEN
```

CSE, IIT Kharagpur

6

And then if we find that cm has decreased- if cm has remained the same, then there is nothing else to do. It is already there in open or closed and the new path that we have discovered, is a larger path, so we just ignore it. But if the cost has decreased and m is already in closed, then we will move it to open. If m is in open, then we just update the cost and leave it like that, because then, subsequently, m will be expanded, but if m is not has already been expanded and has been taken to close, then what has happened is that we have found a better path now. And so not only do we have to again expand m, but we may have to expand several successors of m also, because their cost may also have decreased, right? And then finally-actually, here I have left out the last step. The last step is then, again, go back to step 2. That is the loop, right?

Let us take an example to work this out and see what happens. Then, we will compare this with some of the algorithms that we already know and you will see that there is a lot of familiarity here. Let us take one state space which has the following states- let us say one. Can you see this size from the back? Let us say that this is the goal state. This state space will actually be given to the algorithm in an implicit form. So, I am just drawing this for our convenience. Actually, the search algorithm will not be able to see this whole graph at one time; it will only be able to see the nodes that it expands. Let us say that the costs are as follows: this is cost 2. We will start by maintaining 2 lists: one will be our open and the other will be the closed list.

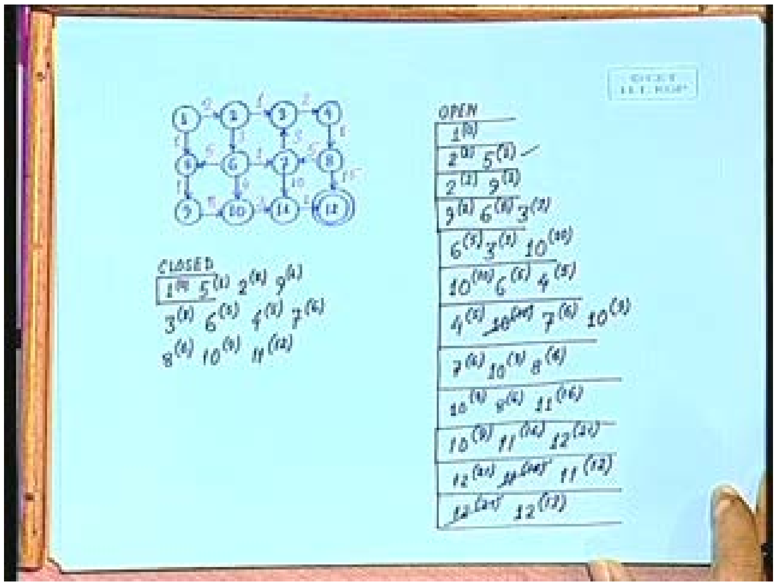
Initially, open contains the state one, and we will associate the cost of 0 with it. Remember, in the first step of the algorithm, we had set cs is equal to 0, right? So, we will expand open in the first step and put open in put one in closed. We will put this in close. And what is open going to contain now? The 2 successors- 2 with a cost of 2, and 5 with a cost of one, right? Then, we are going to pick up the minimum cost state from open. So, we will pick up 5 and then, we will put 5 in closed, with cost one. And in open, we will now have- 2 will remain with a cost of 2; 5 is going to get expanded. So, we will have 9 with a cost of 2, right?

Then again, out of these 2, these 2 have the same cost. So, we will pick any one of them. Let us say that we pick up 2. Then, we will have 2 with a cost of 2 in closed, right? And then, here, we will have 9 with a cost of 2. And for 2, we will have 6 and 3, right? So, 6 with a cost of 5 and 3 with a cost of 3. Okay. Then, next step: we will be picking up 9 with a cost of 2. And what do we have here for 9? Only 10, right? So, 6 with a cost of 5 will be there. 3 with a cost of 3 will be there and we will have 10 with a cost of 10, right? Then, we will be picking up 3. Next, in closed, we will put 3 with a cost of 3. What will we have here? We will have 10 with a cost of 10, 6 with a cost of 5 and we will have 4 with a cost of 5. Again, we have a tie. Let us say that we pick up 6. So, we take 6 with a cost of 5.

Now, something interesting will happen. When we have 6 with a cost of 5, then okay. We will have 4 still, with a cost of 5, right? We had 10 with a cost of 10. Now when we expand 6, we are going to have 2 new entries. One is 7 with a cost of 6, and now 10 comes with a cost of 9. It is already there in open. So, we just update the cost and we now have 10 with a cost of 9. If it is already in open, we just update the cost. See, so far, nothing has come out from closed to open. Nothing has come out, right? Then, what will we pick up next? 4 with a cost of 5 will come out. So, we will have 7 with a cost of 6, 10 with a cost of 9, 8 with a cost of 6, right? Then, let us say we pick up 7 with of 6. What do we have here? We have 10 with a cost of 9, 8 with a cost of 6 and for 7, we will have 11 with a cost of 16. Then, we will pick up 8 with a cost of 6 and we will have 10 with a cost of 9, 11 with a cost of 16, and we now have 12 with a cost of 21. So, the goal has come into open.

Goal has come into open, but if we just pick up the goal with a cost of 21, we will be incorrect. We still cannot pick up the goal, though it is there in open. So, what- because there might still be a better path to reach this goal, and because we are optimizing and our objective is not just to find the goal, but to find the best path to the goal, we will not stop here. We will continue doing this, until we pick up the goal for expansion. Next, what will we pick up? We will pick up 10 with a cost of 9. That is going to give us 10 with a cost of 9. Then, we will have 12 with a cost of 21. We had 11with a cost of 16, but now when we expand 10, we get 11 with a cost of 12. We replace this with 11with a cost of 12, right, and then we will pick up 11 with a cost of 12. And we have a cost of 21, but this is going to get updated because of the expansion of 11. And we are going to get 12 with a cost of 13, right?

(Refer Slide Time: 25:41)



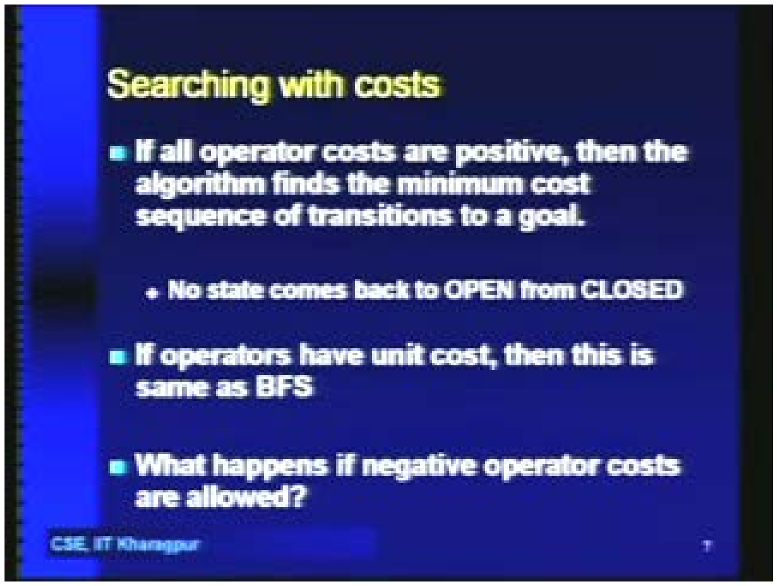
Next, 12 is selected for expansion. That is where we terminate. So, we find that the path- the best path to the goal node has a cost of 13. (Student speaking.) We will come exactly to this analysis now. We have to be now be sure that what we have done is correct, right? Yes. (Student speaking). The algorithm, yes. Slide. (Student speaking).Yes. (Student speaking).It is possible. We will come to all those cases now. We will systematically explore all this cases. It will depend on the kind of state space that we have. Now, do you see familiarity of this algorithm with something? Dijkstra's algorithm. This- when we are looking at positive edge cost. When we are looking at positive edge cost, this is nothing but Dijkstra's algorithm, right? But, when we have costs with negative cost, then we know that Dijkstra's will not work. And the reason is exactly as you were asking- that node can come back from close to open, because even after you have expanded that node, you might find another path to some other node of higher cost.

In this case, what is happening is, if all the edge costs are positive, then, when you are expanding your nodes, you know that there cannot be any other node which has lesser cost, because you have already explored the frontier up to that cost. And any nodes beyond that frontier is going to add up more cost with it- more positive cost with it- so those other states are going to have more cost. So, you cannot have anything with lesser cost, right? But, if you allow negative edge costs, then it is possible, that a successor of a node of cost 20 has a cost 5. If you have expanded the cost up to frontier of 20, they could still be some successor down below, which has a cost less than 20, because of the negative cost, right?

In those cases, we may have to bring back the node from close to open and it will still work. It will not work, however, if you have a negative cost cycle. Yes. (Student speaking). Where will we stop in that case? We will stop when we have exhausted open. Now, can it be possible that open is never exhausted? If you have a negative cost cycle, then, open will never be exhausted. You will keep on picking up nodes from open and putting it back into open after finding a better one, because in that loop, the cost of the nodes will keep on decreasing in the negative cost cycle. So, you will keep on moving nodes back from close to open and we expand in them, right, but as long as you do not have a negative cost cycle, then, your algorithm will terminate, right?

So, result number one is, if all operator costs are positive, then the algorithm finds the minimum cost sequence of transition to a goal. And this is exactly as in Dijkstra's: if operator costs are positive, then, what we have is exactly Dijkstra's and no state will come back from closed to open. If operators have unit cost, then, this is the same as breadth first search. Yes or no? If all operators have unit costs, then, the cost of a node is the minimum depth at which it can be reached, right? So, you will expand it in that sequence. You will be expanding the nodes at a depth of 1, first, the depth of 2. We are going breadth first, right? And then, we have already discussed this. What happens if negative operator costs are allowed?

(Refer Slide Time: 30:41)



Searching with costs

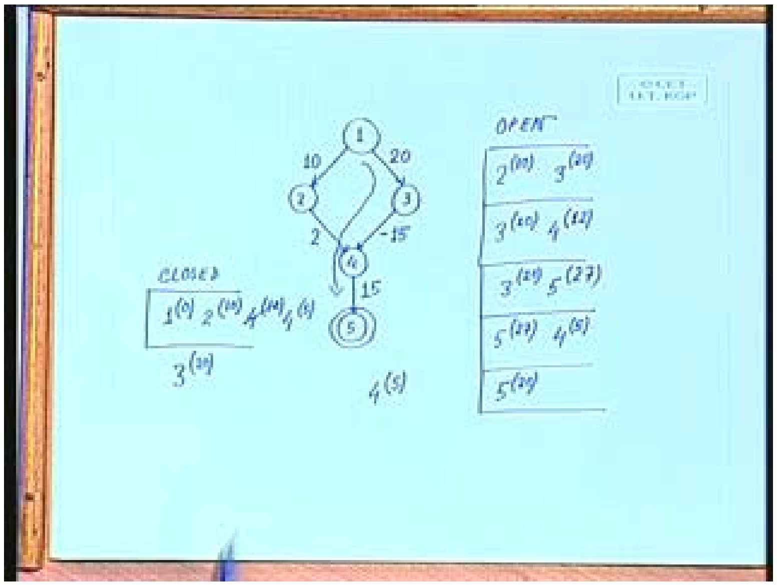
- If all operator costs are positive, then the algorithm finds the minimum cost sequence of transitions to a goal.
  - No state comes back to OPEN from CLOSED
- If operators have unit cost, then this is same as BFS
- What happens if negative operator costs are allowed?

CSE, IIT Kharagpur 7

Can you create an example, where a node is going to come back from closed to open? Try to create an example where a node comes back from close to open. (Student speaking). I am creating a much simpler example, just to demonstrate the point. See, just check out this one. Do you think that this is going to work? Let us see. We will start with 1, right? When we expand this, we will have 2 with a cost of 10 and 3 with a cost of 20, right? And then when we expand 2, we will get- okay, let me increase this cost; increase it to something like 15, right? So, when we expand 2, next thing that we are going to have is, we will have 3 with a cost of 20 here, and we will have 4 with a cost of 12, right?

So, what do we have in open? What do we have in closed? In closed- this is open. In closed, we have 1 with cost of 0 and now we have put 2 with a cost of 10. Next, we are going to pick up 4 with a cost of 12 and that is going to give us 3 with a cost of 20 and 5 with a cost of- how much?- 27, right? Even if this is a goal, we cannot pick it up right now. So, we will pick up 3 with a cost of 20. We will have 3 with a cost of 20, and that is going to give us 4. We will have 5 with a cost of 27 and now we have created 4 with a cost of 20 minus 15. So, 5. Now, it is in closed, but we have reached it with a lesser cost. We will bring it back into open with a cost of 5, right, and then, we will expand we expand this node, okay? This is now 4 with a cost of 5, will come back here after the expansion. And what are we going to have here? We are going to have 5 with a cost of 20 and then when we pick this up, we have discovered the least cost path to the goal, which is of cost 20, along this path, right?

(Refer Slide Time: 35:03)



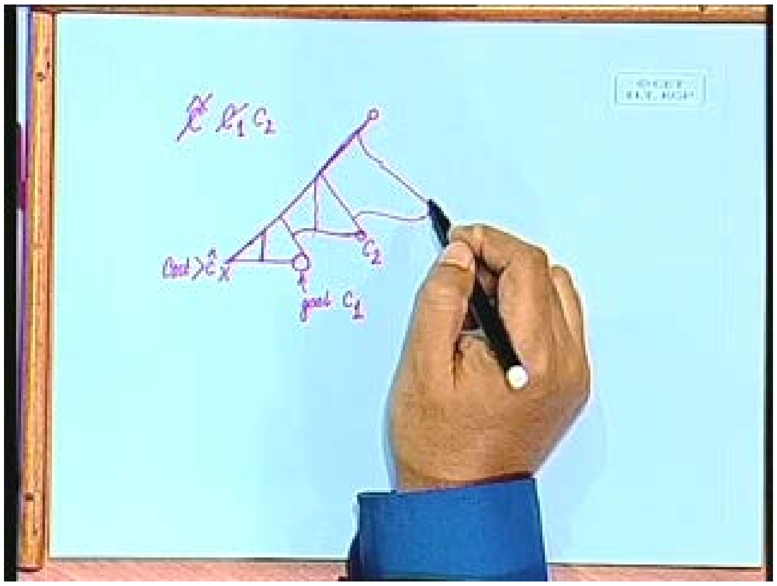
Now, let us- okay. Now, we are going to study another algorithm, which also is very useful. This is called branch and bound- least- or, in other words, what we will study here is again another family of algorithms, called branch and bound. In branch and bound, what we are going to have is, we will start by knowing an upper bound on the solution cost, right, and we are going to proceed something like: we can proceed depth first. We will proceed depth first, find out one goal and whenever we find out one goal, we still do not know that this is the least cost goal. But we can use this as an upper bound, to prune out other parts of the search space, right?

In other words, what we are going to do is, start with some bound. Let us call it, say c hat. Say this is the bound that we start with, and then, I start doing a depth first traversal from along the state space. I continue, until I reach a state where the cost exceeds c hat. Whenever that happens, I backtrack and then try some other operators, right? In this way, I am going to follow the cut off point of c hat, right, until I find, along some path- I reach a goal. Suppose I find a goal and let us say that this goal has cost of c1. Then, what I am going to do is, I am going to replace c hat by c1, and now continue the search with c1 as cut off. So, any path which costs more than c1 will be pruned. I will not visit any further. I will backtrack until I find another goal which, say, has cost c2. Then my bound will become c2 and I will again do a depth first traversal with the new frontier as c2, right?

So, c1, c2, c3- like that, I will progressively refine my upper bound, until I am finished with the entire search space. As long as I am not finished with the entire search space, I cannot say that there is no better cost. When I have finished searching the entire search space with these bounds, then, my current bound gives me the cost of the minimum cost solution. Is it clear? Will you be able to write a program for depth first branch and bound? Not difficult, recursively write the thing. Only, at every step, you check whether the current bound has been exceeded or not. Suppose you are asked to write the traveling salesperson problem using depth first branch and bound. You should be able to write it. You start with one city, then, recursively traverse all the remaining cities. And when you have first found the first tour, take any tour and you just find the first tour. Then, that tour becomes your upper bound, right?

There are, However, the performance of branch and bound will depend, to a large extent, on the first upper bounds that you derive. See, this complexity of whole thing will depend on how quickly you are able to find out a goal whose cost is merely as close as the optimal goal. If you can quickly find that, then you have a tight upper bound and then your search complexity will be less, right?

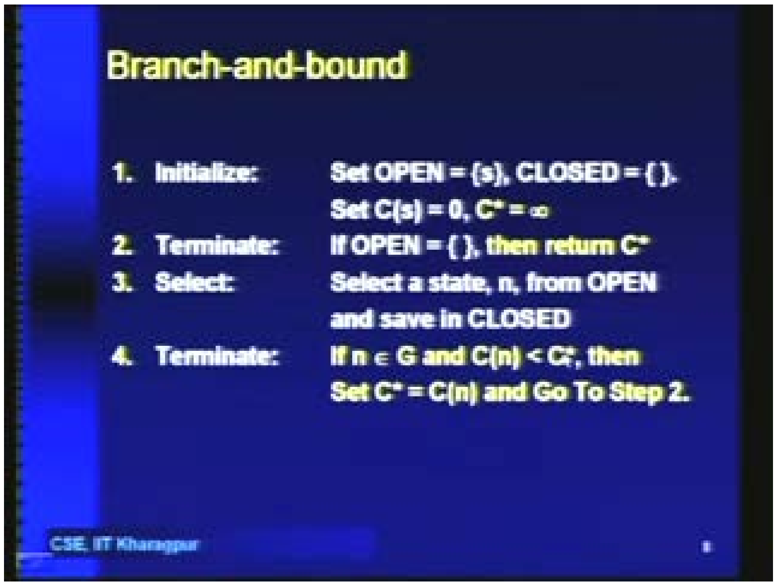
(Refer Slide Time: 39:39)



So, to do that, there are different heuristics. There are heuristics which will- or greedy algorithms, which will give you a pretty close approximation of the cost. What people do is, they first apply a greedy approximation algorithm, get a reasonably tight upper bound on the solution cost; then, they use that solution cost as the upper bound for doing depth first branch algorithm. So, if you cast branch and bound in the same framework as we have done so far, then this is what we are doing. We are maintaining this cost C*, right? This C* is the bound, so initially, if you do not have any estimate, then you treat C* as infinity. Then, when open is empty, then return C*. As I said, that you have to exhaust the search space; you have to finish off all states in the search space and only then, you can terminate.

So, we will return C*, or the current cost, when we find that open is empty, right? Otherwise, we just move as previously: we select a state n from open and save it in closed. Then here, the terminate step is different. What we do here is, if we find that n belongs to g, which means n is a goal, and the current cost that we have found of n is less than C*, That means we have found a better goal; we have found a goal having lesser cost. Then we update C* to this new goal cost and then go to step 2, which means that we again keep on exploring, right, but so far, we have not used the bound C* to prune anything.

(Refer Slide Time: 41:53)



Branch-and-bound

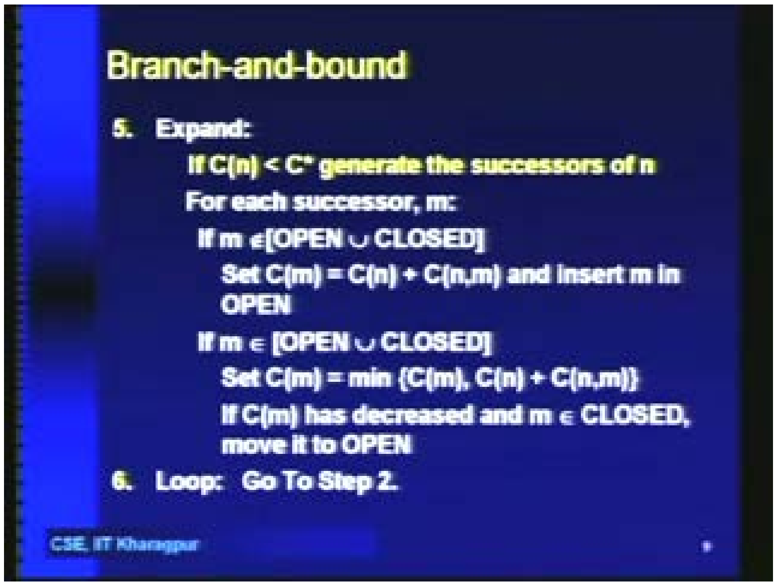
1.	Initialize:	Set OPEN = {s}, CLOSED = { }. Set $C(s) = 0$, $C^* = \infty$
2.	Terminate:	If OPEN = { }, then return C^*
3.	Select:	Select a state, n, from OPEN and save in CLOSED
4.	Terminate:	If $n \in G$ and $C(n) < C^*$, then Set $C^* = C(n)$ and Go To Step 2.

CSE, IIT Kharagpur

Let us see what we do in expand. This part is clear, right? Let us see what we do in expand. In expand, what we do is: we check whether the cost cn is less than C*. If c n is not less than C*, then there is no point in generating the successors of n, because you are not going to get a better cost goal than this, right, assuming that all costs are positive. Then, for each successor m, we do exactly as before: that if this happens, this happens. Finally, we go back to step 2. So, it is the same as above, except that we will expand a node, only if we find that it has a promise of giving us a better goal.

So, if its cost has already exceeded, or has equaled the cost of C*, then, there is no point exploring it further. And we will just- but if you have negative edge costs- then, this will not work, because you will not be able to prune it. You will not be able to prune it, because you can still have some node which has better cost, better goal cost than this. Yes. Therefore, in those cases, we will not be able to use this branch and bound. At least this kind of branch and bound. There are other algorithms, however, but this basic branch and bound we cannot use.

(Refer Slide Time: 43:32)



Branch-and-bound

```
5. Expand:
      If C(n) < C* generate the successors of n
      For each successor, m:
        If m ∉ [OPEN ∪ CLOSED]
          Set C(m) = C(n) + C(n,m) and insert m in
          OPEN
        If m ∈ [OPEN ∪ CLOSED]
          Set C(m) = min {C(m), C(n) + C(n,m)}
          If C(m) has decreased and m ∈ CLOSED,
          move it to OPEN
6. Loop: Go To Step 2.
```

CSE, IIT Kharagpur

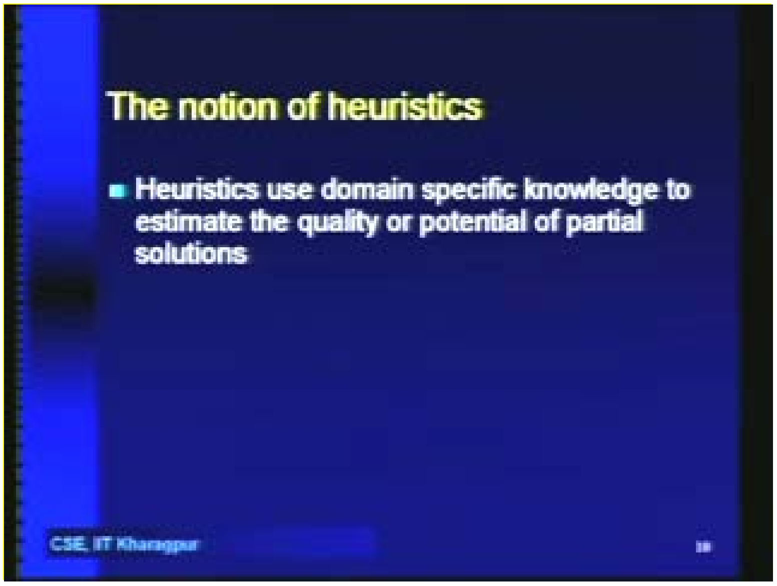
Now, can you tell us that what makes us choose between the algorithms that we have studied so far? We have studied basic DFS, BFS, then uniform cost search. We just searched like in Dijkstra's and we have also studied branch and bound, depth first branch and bound. Which do you think will work best in a for a problem which has a large state space? Okay. So, there are several things that we need to consider: one is, we need to consider the amount of branching that we have. That is one factor. There are some state spaces which are shallow, but which have a lot of breadth. If you take the travelling salesman problem, say, if you take something like fifty cities, the depth is only fifty and that is not much. But, if you look at the size of the state space- because, from every city from the start city, you have 49 different options.

From the second level you have, right. It grows in a sort of factorial way. That is why, it is not it is the breadth that you have to contain. If you try something like BSBB, then it is more likely to work nicely, because if you get a reasonably tight upper bound- and we will see some heuristics for doing that- then you can work in a recursive way. Your space requirement will be less. You do not have to store the breadth, because you are going depth first, right? So, if you use DSBB or if you use iterative deepening also- will iterative deepening be good for DSB? Yes or no? You remember iterative deepening? We studied in the last class. What we are doing is, we are progressively doing DFS with increasing bound. See, the problem here is that all your solutions are at the same depth, so there is no point in doing iterative deepening here.

Depth first branch and bound will work, because in many tours, we will be able to prune off early, because the cost already exists there, right? If you do something like breadth first search, the amount of branching that we will have will kill you. On the other hand, there are state spaces where the depth is pretty large, but the breadth is not much. That will happen if there are many- if you have state spaces, where you keep coming back to the same set of states along different paths. It goes like this: you start with a set of states, then, it just grows a little bit and then again converges to some state, and again grows a little bit and converges to some state.

You have state spaces like that, then the branching factor is not much, but it can go up to a long depth, right? There, you may have a goal which is pretty close to the start state and if you go depth first, you will be doing a lot of work, right? As we do some problems in the lab, which we will do from in another couple of classes, I will give some assignments to work out in the lab. You can see for yourself by characterizing these things out that what works best. So, in this, let us first just move. I will introduce one or 2 more concepts and then close this particular lecture. What we are going to leave off with this lecture with, is the notion of heuristics. Now, what is a heuristic function? Heuristics are domain specific knowledge, to the estimate the quality or potential of partial solutions.

(Refer Slide Time: 48:40)



The notion of heuristics

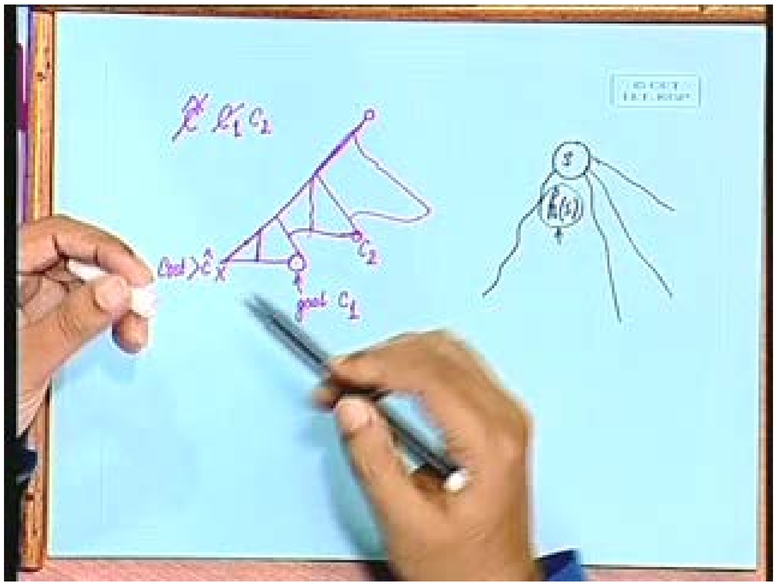
- Heuristics use domain specific knowledge to estimate the quality or potential of partial solutions

CSE, IIT Kharagpur

19

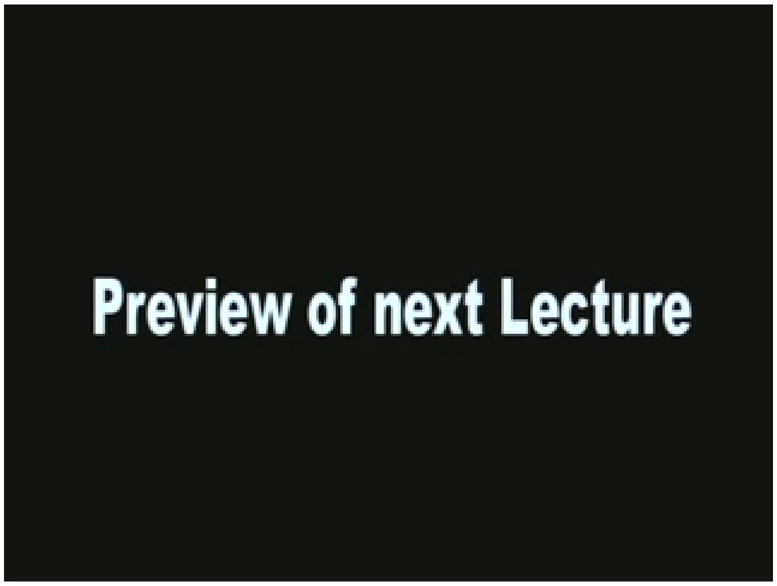
What we are expecting here is that, when we are in a particular state of the state space, and suppose we know what the goals are, that we want to reach. Then the heuristic function at a state s, will be a function- let us call it hs. It is going to give us an estimate of the cost of going from s to a goal. it is We do not know exactly where the goal is, but it could be at any depth, anywhere. But this gives us an estimate of the cost of reaching the goal, right? So, this estimate can be an underestimate, it can be an overestimate, right? We will see that in the cases where this is an underestimate, we have a family of algorithms, which will work pretty nicely. And then we will see that if the heuristics overestimate, then the same algorithms can perform well under certain circumstances and in other cases, it may not terminate at all. So, in the next lecture, I will start introducing the notion of heuristics.

(Refer Slide Time: 50:03)



I will also talk about the heuristics used in certain known problems and then, we will see how algorithms can utilize these heuristics to solve the problem in a better way.

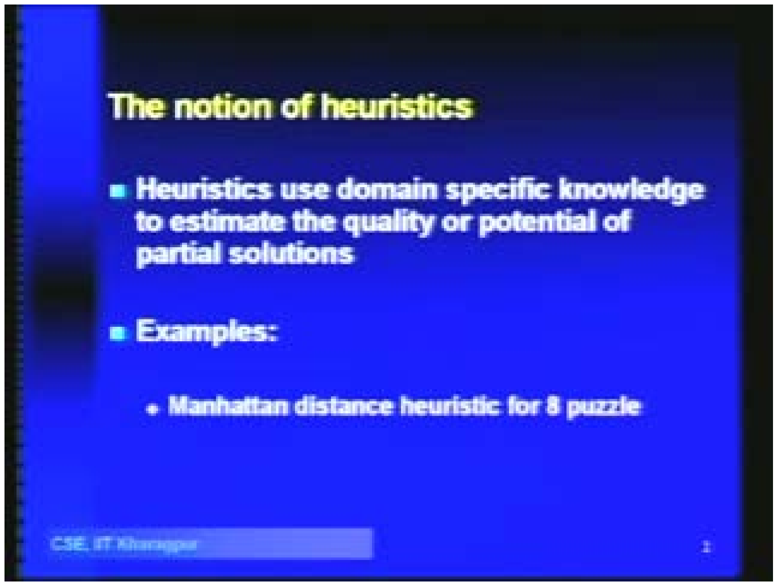
(Refer Slide Time: 50:16)



Preview of next Lecture

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 I 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 few examples. The one of most common heuristics for the 8 puzzle is the Manhattan distance heuristics.

(Refer Slide Time: 51:48)



The notion of heuristics

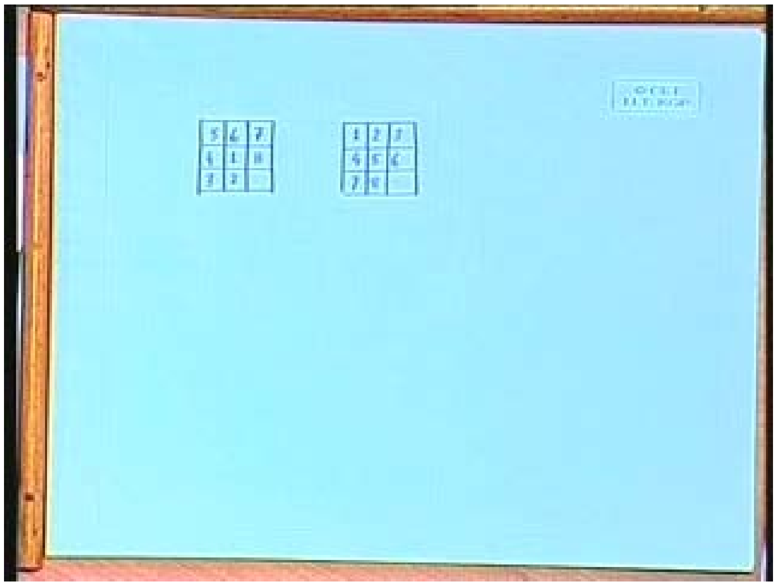
- Heuristics use domain specific knowledge to estimate the quality or potential of partial solutions
- Examples:
  - Manhattan distance heuristic for 8 puzzle

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. 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: 52:48)

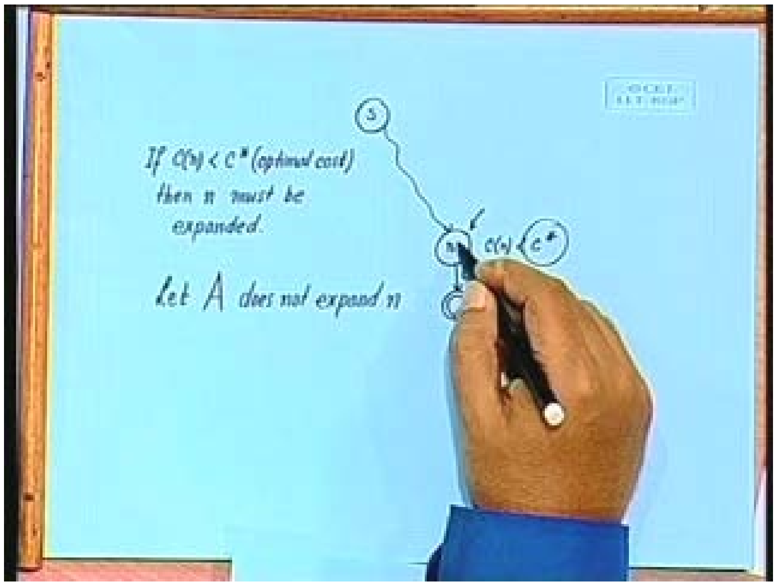


So, to travel from place c to place b, the total distance that you have to travel is your distance northwards plus your distance eastwards or westwards, right? You know why? Because suppose there is some state. This is our start state. And there is some state here; some state n. And the cost of n the 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 it does not expand n, then it will never discover this goal and 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, okay? Let algorithm a does not expand n. Then, what we can do is, we can keep the remaining state space identical.

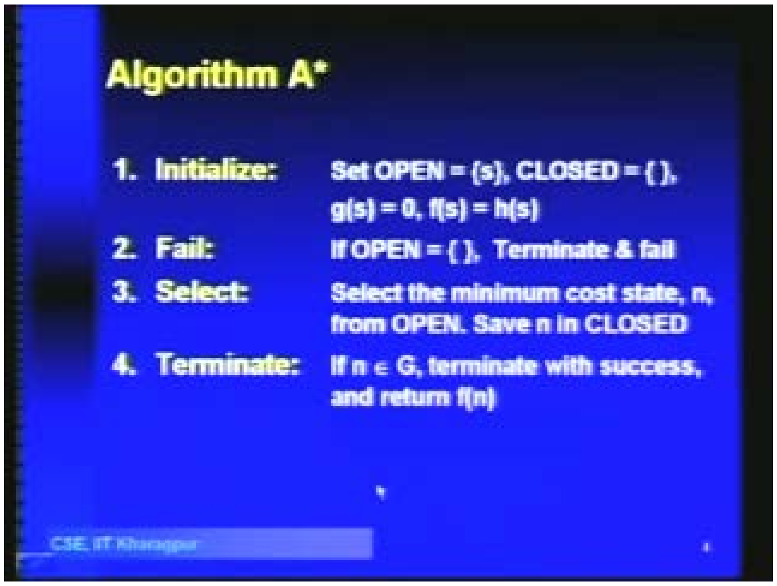
You do not make any change to the remaining state space, except that below n, we just add a goal and give it a cost which is between cn and C*. We can always do that, because cn is less than C*. You can have some epsilon which you add to cn, and then this edge cost is that epsilon, right? This goal will 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 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.

(Refer Slide Time: 57:44)



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 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 the accuracy of the heuristics, right? Okay.

(Refer Slide Time: 58:35)



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

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 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 close, and it uses up too much of memory. So, it does not work well in practice, but there are variants of that which are used. We will study some of those in the next lecture.