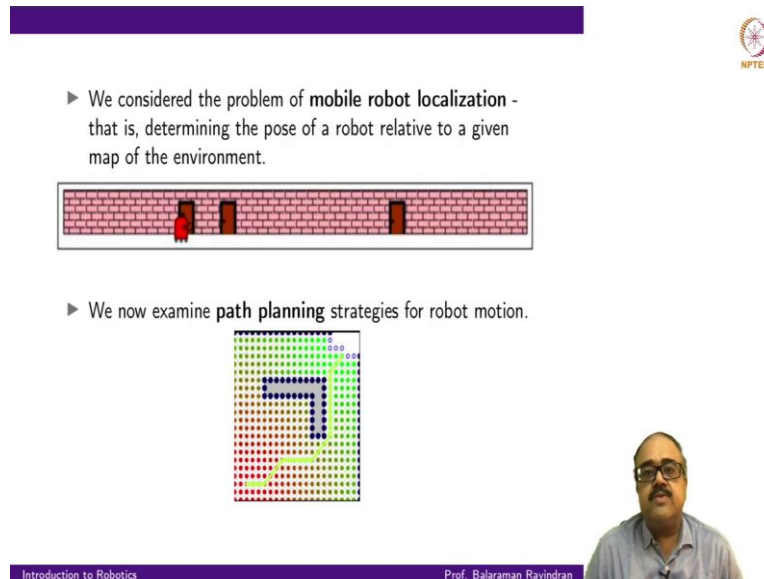


Introduction to Robotics
Professor. Balaraman Ravindiran
Department of Computer Science
Indian Institute of Technology, Madras
Lecture No. 43
Path Planning

(Refer Slide Time: 0:13)



The slide features a purple header bar at the top. On the right side, there is the IIT Madras logo. The main content consists of two bullet points. The first bullet point is followed by a diagram of a 1D environment with a red robot and two obstacles. The second bullet point is followed by a 2D grid-based path planning diagram showing a path from a start point to a goal point around obstacles. At the bottom right, there is a small video inset of the professor. A purple footer bar contains the text 'Introduction to Robotics' and 'Prof. Balaraman Ravindiran'.

- ▶ We considered the problem of **mobile robot localization** - that is, determining the pose of a robot relative to a given map of the environment.
- ▶ We now examine **path planning** strategies for robot motion.

Hello everyone, so this week so far, we considered the problem of mobile robot localization, we looked at the localization taxonomy. We also looked at the Markov localization. And we saw a little bit on how to do this localization, given a feature base map, most of the algorithms are looking at grid based map.

Now, for the next lecture, we are going to look at path planning strategies for robot motion. So, what does this path planning strategies mean? The path planning strategies means given a map, and a localization algorithm, so figure out a path, that is feasible, so that I can get from some starting location, let us say here, to some ending location, which is say in the top corner of the robot, there is some obstacle here in the middle. So, all of this is known to me in the map. And my goal is to find a path, a feasible path, starting from a given location, and going all the way to the destination, which here is in the top corner.

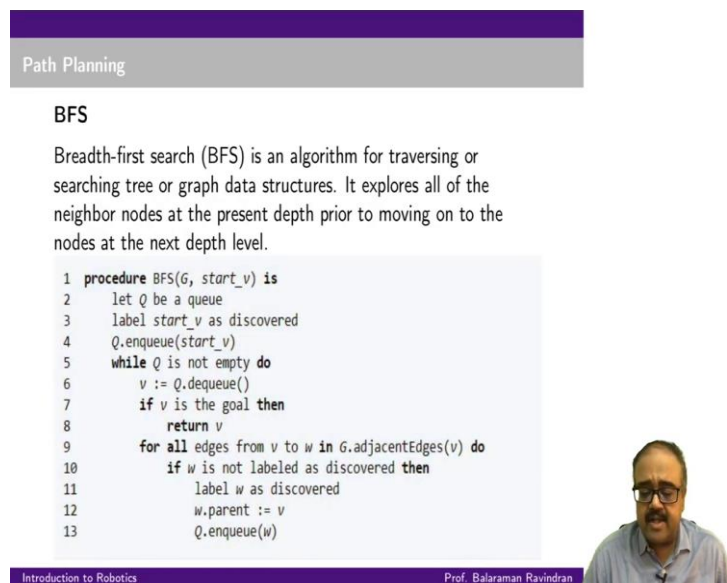
So here, we are going to assume that the map is given to us, and whatever discussion I am doing will be in the context of location-based map, like a grid base map or anything. But then I could think of other kinds of strategies as well. But the thing with path planning is, it is good to know the open space as well as the occupied space in the in the world, so in the map, so that makes it easier for me to plan a path, a feasible path would always take my robot

through open space, you must have seen this in sea space, how we look at avoiding collisions and things like that. So, feasible paths will do that.

So, I am going to look at strategies for searching for feasible paths. It happens that the algorithms that we will look at all actually look at optimal paths. Basically, they return the shortest path to go from one location to another. And you can think of other algorithms wherein more complex environments, we are looking at all possible pathways are not really that feasible, even examining a good fraction of the pathways are not feasible.

There are more probabilistic path planning algorithms, which with high probability give you a good path. But they will return the feasible path. So those are not, those are for the future. So, for things that you could look at later.

(Refer Slide Time: 2:41)



Path Planning

BFS

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

```
1 procedure BFS(G, start_v) is
2   let Q be a queue
3   label start_v as discovered
4   Q.enqueue(start_v)
5   while Q is not empty do
6     v := Q.dequeue()
7     if v is the goal then
8       return v
9     for all edges from v to w in G.adjacentEdges(v) do
10      if w is not labeled as discovered then
11        label w as discovered
12        w.parent := v
13        Q.enqueue(w)
```

Introduction to Robotics Prof. Balaraman Ravindran

So, we will start off with a very simple path planning algorithm called a breadth-first search. Breadth-first search is an algorithm that was originally designed for traversing any kind of graph data structure, starting from a node. So, in our case, the graph data comes from the connectivity in the map. Suppose this is like my grid map. And let us say that each one of these squares is a grid. So, I can say that each grid cell, is connected to 8 of its neighbors. So, you could think of this as a graph, where each node has 8 neighbors.

So, some nodes will have only 3 neighbors, some will have fewer, so this one has only 5 neighbors. This node has only 5 neighbors, and this node has only 3. And so, nodes that are here actually have neighbors that are not reachable, so like that, so you can think of this free space here as a graph. And so, breadth-first search allows you to search for a path on this

graph, so you can think of that each node has 8 neighbors, mostly. And then I have one node, which is the start node for me, which is here, and one node that is the end node, which is there, and I am going to do breadth-first search. Basically, I am going to search this graph till I get to the code.

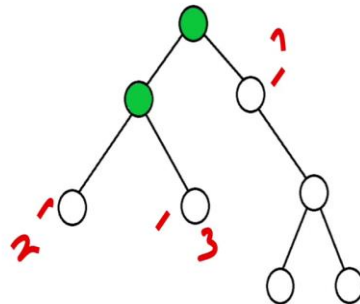
So, here is how breadth-first search will operate. So, given it starts vertex v , and a graph, G , so in this case, the graph would be a representation of the free space of the map. So basically, it starts with v , it is, it puts v into some Q , so it is some kind of data structure Q . And as long as there are nodes there are in Q . As long as Q is not empty, it is going to take the first element from the Q , that is what the dQ means, it will take the first element and if v is the goal, then say return hey, I found the goal. But if v is not the goal, then we look at all edges that go out of V .

So, what you mean by all edges had got of v ? So, in this case, it will be all the 9, all the 8 neighbors, so take a node, so it will be all the 8 neighbors that are there for that node v . For all the edges that go to v , then if w has, so you call this vertex w , so if w is already not labeled as discovered, then label was discovered, say hey, now I have seen w , if it is already been seen, ignore it. But if it is not been seen before, then I will say, now I have seen w , and I maintain this parent data structure, so that I can actually recover the path and the once I reached the goal. And then I will say that the parent of w was v .

So, what does this mean? That w was visited for the first time, when I expanded the graph from v . Now, I put w in the Q , and I keep going. So, when will Q be empty, either, so when will this loop end? Basically, the loop will end, if I find the goal, or I have explored all the vertices in the graph and there are no more vertices for me to see. We explored all the vertices in the graph, and there are no more vertices for me to see.

So, I will be saying vertex and node, exchangeably, but to denote a cell in the graph. So, each cell here, each cell in the free space becomes a node in the graph. And if there are the adjacent cells to that are connected by an edge in the graph, so I will say sometimes node, I will say vertex, it is a common terminology in computer science, I will go back and forth.

(Refer Slide Time: 6:44)



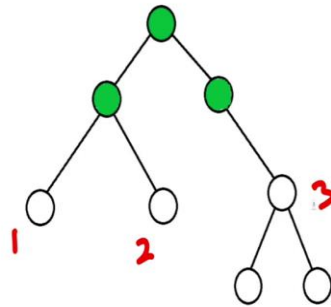
So, let us look at pictorially, how this is going to look like. So, here is my graph. So, you can think of this as a place where each cell has either 2 or 3 neighbors, 1, 2 or 3 neighbors. So, this is a map in which each cell has 1, 2 or 3 neighbors, I am starting from the cell, marked green, so that is my start node. Then, what I do is I first visit the node to the right and then I mark that as visited, then I will add these 2 nodes to the Q.

So, first, when I visit this, I will add this and this to the Q. Let me pick up. So, when I first visit this node, I am going to put this in the Q and this in the Q both. And I am going to put this in the Q and this in the Q, these 2 nodes, go into the Q. Then what I do, I look at the first node in the Q, which is this so I will I will pick up. So, I will pick up this node in the Q, as the first node in the Q, and then I will add this node, and this node to the Q.

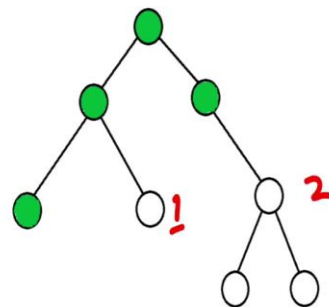
But remember, this is already there in the Q. So, that is the number 1, so this will be number 1 in the Q. This will be 2, this will be 3, and this will be 3. So, I have 1, I have 2 and I have 3. So, what will be the next node that I will take up for expansion?

(Refer Slide Time: 8:15)

Path Planning



Path Planning



Exactly, so this is the node that I marked 1, now what will happen in the Q, I will have this is 1, this will be 2, and that will be 3 in the Q. So, the next node I will take up is the 1 on the bottom left, so the node that I consider, will be that. And now I do not add anything more to the Q. So, my Q will still have this as 1. And that as 2, I will continue with the 1, and that is not going to change much.

(Refer Slide Time: 8:52)

Path Planning

Introduction to Robotics Prof. Balaraman Ravindran

Path Planning

Introduction to Robotics Prof. Balaraman Ravindran

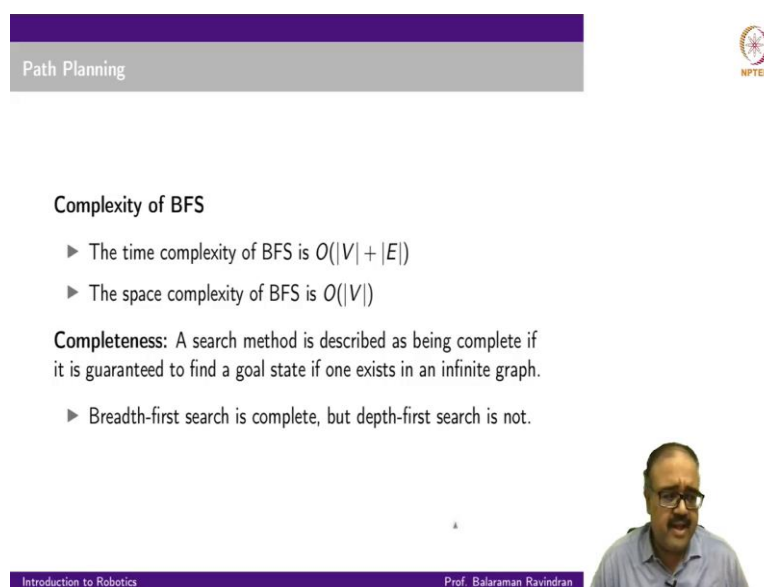
And then I will go to the next one. Now, what happens is, the Q could have become empty. But I am going to add these two to the Q now, and therefore I will continue, I will finish that. And if any of these had been the goal, I would have stopped much before. So, here is just an illustration for how breadth-first search will look at every node in the graph. And if there is a goal somewhere in between before this, I will find it and it is guaranteed.

As long as the edges do not have any weight, as long as the edges are all having the same weight, either they do not have any weight or is another way of saying is they all have the same weight, then I am guaranteed to get to the, get the shortest path to the goal, guaranteed to get the shortest path to the goal. So, how is that? So, suppose you remember, whenever I mark a node as visited, whenever I mark a node in green, I will add a parent to that node.

So, what will be the parent of this node? The parent of this node is that. And, what will be the parent of that node? That is that. Suppose I want to reach from here to here. Now, what I want to look at is I will go to the target node, I will keep looking at the parent of the parent of the parent of the node until I reached the source node. So, that is a way I will reconstruct the path.

Suppose I want the path from here, to here from here, so I will keep going and from this 1, I look at the parent which will be this and for this, I look at the parent which will be this, for this I look at the parent, that is a source node that I wanted. So, given of source node, I can find the shortest path to any target node by just following the parent marking that ended in my graph that ended in my data structure. So, that is basically how this algorithm will operate.

(Refer Slide Time: 10:55)



The slide is titled "Path Planning" and features the NPTEL logo in the top right corner. The main content is under the heading "Complexity of BFS".

Complexity of BFS

- ▶ The time complexity of BFS is $O(|V| + |E|)$
- ▶ The space complexity of BFS is $O(|V|)$

Completeness: A search method is described as being complete if it is guaranteed to find a goal state if one exists in an infinite graph.

- ▶ Breadth-first search is complete, but depth-first search is not.

At the bottom of the slide, there is a video feed of Prof. Balaraman Ravindran. The footer of the slide contains the text "Introduction to Robotics" and "Prof. Balaraman Ravindran".

And so, it looks at every vertex and every edge at least once and does not look at it more than once, every vertex it looks at only once and every edge it looks at only once. And therefore, the time complexity is order of plus V plus E, and it needs to store the pointers and other things, the parent pointer and other things for every vertex like whether it has been visited and also the parents. So therefore, it requires storage of the order of V.

And, so BFS is a complete method, so a method is described as complete. If it is guaranteed to find a goal state, if the path exists, if there is a path from the start state to the goal state, then a complete algorithm is always guaranteed to find it. And breadth-first search is complete. But an equal algorithm called depth-first search, where I just keep going. So, if I

find a node has a child, I will just go to the child first. So, I will not go to the other nodes in the Q. So, if the node has a child, I will keep going to the child. And then after I finish with the child, I will come back, and I will see if there is another child to the node.

If you think of the graph this way, so instead of, so if you think of breadth-first search, it went in this order, depth-first search would go in this order. So, this node will be looked at, will be marked as visited only after this node is completed. In fact, this node will be marked as visited only after this node is completed. So, depth-first search could get stuck exploring a wrong path. If it is arbitrarily deep, when the path is infinitely long, then it will never come back and find a goal that is here.

But breadth-first search will find a goal that it is a finite depth, even if the graph is infinitely large. That exists a path breadth-first search will find it. So, that is why we, what we mean by saying breadth-first search is a complete algorithm. But the problem in breadth-first search is it expands a lot of nodes, it visits a lot of nodes looks at neighbors of a lot of nodes, before it moves on to the next, it expands a lot of nodes, for every node, it expands all its children, it looks at all the children puts them in the Q before it moves on to the next level.

And so, for every level, it is going to expand the level children of all the nodes at the previous level, even if it is very clear that it is not going to give a short path. So, this is a little tricky, especially when I start having weights on the edges, because all edges are not the same anymore. And I really like to follow edges that are more likely to give me optimal paths and keep the edges that are less likely to give me optimal paths for later. So, for this kind of weighted graphs, at where the edges have some cost. So, people look at other algorithms.

(Refer Slide Time: 14:02)



Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller.



So, the most popular of this is something called the Dijkstra's algorithm. So, the Dijkstra's algorithm is essentially, it finds the shortest path between any 2 vertices a and b in a graph. And this works on graphs that have weights, I mean from a, if you want to be more picky about it, it works when graphs have positive weights, and not negative weights. But then we are talking about path planning here, so paths have positive cost, they are not going to have negative cost for us.

And therefore, Dijkstra's algorithm works perfectly alright. And, so among all the vertices that I have not visited so far, Dijkstra's picks the vertex with the lowest distance, and calculates the distance from that vertex to each of the unvisited neighbors of that vertex. And updates the neighbors in distance if it is smaller, so we will just see how it works.

(Refer Slide Time: 14:56)

Path Planning



```
1 function Dijkstra(Graph, source):
2
3   create vertex set Q
4
5   for each vertex v in Graph:
6     dist[v] ← INFINITY
7     prev[v] ← UNDEFINED
8     add v to Q
9   dist[source] ← 0
10
11  while Q is not empty:
12    u ← vertex in Q with min dist[u]
13
14    remove u from Q
15
16    for each neighbor v of u: // only v that are still in Q
17      alt ← dist[u] + length(u, v)
18      if alt < dist[v]:
19        dist[v] ← alt
20        prev[v] ← u
21
22  return dist[], prev[]
```



So, let us look at the algorithm here, at any point of time we start off with full graph, and so we have the vertex at Q, which has to be examined, just like we had the Q earlier for BFS, we have the Q. And for every vertex, we start off by saying the distance from the source to that vertex is infinity, because I do not know, I have not found out the distance from the source to the vertex. So, it might as well be infinity.

So, I am just putting it at a very large distance, infinity means a very large number. And of course, I cannot represent infinity itself, it means that probably the largest number that I can represent in my computer or whatever. And then I say that the previous node in the graph, like we had the parent in BFS, the previous thing is now starts off with undefined because I do not know how to reach to B.

And, now I put all the vertices in Q, so I need all the vertices in Q. And then at the end of it, the distance to the source is 0, because I know how to get to the source, I am starting there. And then just like we had in BFS, I will say with Q is not empty, so, I will pick the vertex in Q that has the minimum distance u. Remember what is distance, the distance u is the distance of the vertex u from the source.

How much, what is the cost of the path, starting from source to get to u is what I mark as distance u, now notice that at any point in the algorithm, the distance u might not be the correct shortest distance, but at the end of the algorithm, the distance u will be the shortest distance from the source. So, now I, so what do I find now, first, I pick the u that has the

shortest distance so far, that I have estimated from the source, and I will take the u and examine this u , now u is going to become visited.

So, what I will do is? I will take the u from Q , and for each neighbor of u , so I look at how far I have come from the source to reach u , and how much more I have to go to reach v from u . So, I will compute that. So, I call that as the current alternate distance. If the alternate distance that I have computed is already less than the distance I already have stored for v , remember I start off with infinity.

But it is possible that from some other u prime, I would have visited v already from some other u prime, I could have found out a shorter path than infinity to v . But now what is going to happen is, if this alternate length I computed is shorter than the original length that I had for v , original distance I had for v , then I will change that. I will also change the parent of v to u , because this is the one that gives me the shortest distance. And then finally, after this algorithm completes, I will return distance and that gives me the shortest path length from every, from the source node to every vertex in the graph, that is what distance gives me.

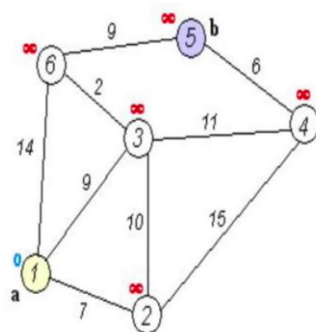
And what is previous gives me? For every node in the graph, previous tells me what is the previous node in the shortest path from the source to that node. Suppose I want to look at previous of u , previous of u will tell me, what is the shortest path, I mean, what is the node just before u in the shortest path from the source to u ? So, that is basically what these things are. So again, this little confusing.

(Refer Slide Time: 18:40)

Path Planning



Illustration of Dijkstra's algorithm



So, let us look at it in a pictorial form. So, I start off with a as my source, so the vertices are labeled 1, 2, 3, 4, 6. And I start with 1 as my source vertex, and 5 as my destination vertex, number I was trying to get from a to b, so 1 is my source, 5 is my destination that I really like to get. And, if I can run it on the whole graph, the Dijkstra's actually gives me, it is called the single source shortest path problem. So, I start off from 1, it will give me the shortest path to all the vertices, if I run the algorithm completely.

And remember, I have the full graph, so these numbers, right now that you see are the edge weights. So, 7 here means, to go from 1 to 2, I have a weight of 7, that means I have to spend 7 units, whatever it is, I have to spend 7 units to go from 1 to 2. And likewise, to go from 2 to 4, I have spent 15 units. So, I really like to get to 5 in the shortest cost possible. And it is not immediately just looking at it in a glance, it is not clear what is the shortest path. So, we have to do some computation.

(Refer Slide Time: 19:53)

So, now I start off with 1. And, then what I do is? I look at the neighbors, so what are the neighbors 2, 3 and 6. So, I will update the distance of 2 to 7, I will update the distance of 3 to 9. And, I will update the distance of 6 to 14, because that is the path cost from going from here to there. And the previous cost was infinity, therefore, I can replace it. And for all of this 2, 3 and 6, I will say the previous node is, the previous will be 1.

Now, which is the node I should be looking at next? The 1 that has the least cost so far, so which will be 2, So I look at 2 next, then I look at the distance from 2 to 3, that is 10. So, the total here is 7 plus 10. So, the total to get to 3 from 2 through 2 is 17. So, 17 is more than 9,

therefore I will keep the 9. So, the previous of 3 is still 1, the previous of 2 is 1, but I will keep that distance here as 9, and then I look at next neighbor of 2, I look at 4, so I have a 7 here, then I have a 15 here.

So, the distance of 4 is going to become 22. And the previous of 4 will be 2. Remember that, so previous of 2, 3 and 6 are all still 1, the previous of 4 is going to become 2. So now, the distance of 4 is 22, which is 7 plus 15, great. Now, I am done with 2.

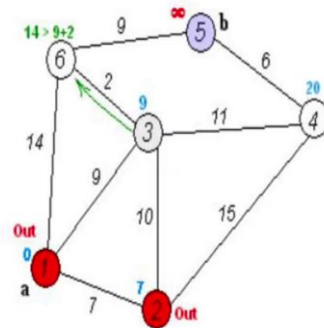
(Refer Slide Time: 21:40)

So, what is the next node I should consider? 3, because 2 has a cost of 22, 3 has a distance of 9 and 6 has a distance of 14. So next node, I will consider is 3. So, 1 and 2 are already been considered. So, I will already complete it, so I will not look at 1 and 2, I look at only 4 as a possibility. So, remember 4 has a distance of 22 already. Now, if I come to 4 through 3, so I will have a distance of 9 plus 11, which is 20, which is less than 22. Therefore, I will replace the distance of 4 as 20.

And, what are the change I will make? I will, so previous of 4, earlier was 2, now previous of 4 will become 3. So previous of 2, 3, and 6 are all 1. And previous of 4 was 2 earlier now it will become 3, because that is the one that I have used for getting that 20 computation. Next, what will I do I look at the neighbor 6. So, what do you think is going to happen?

(Refer Slide Time: 22:57)

Path Planning



So, for 6, it is 9 plus 2, which is, 11. And earlier, we had a distance of 14. Therefore, I will replace that for 6 as well. So, I will make it 11. And not only that, the previous of 6 from 1 will now become 3, the turns out that getting from 1 to 6 through 3 is shorter than taking the path directly from 1 to 6, whatever could be the reason. So, there could be slightly some obstacles here. So, where I have to spend more energy to get through, while going, from going by a 1 by going by a 3, might be a easier path for the robot to move, so whatever reason.

So, it is just an arbitrary graph here. So, I am not assuming any kind of triangle inequality or anything to hold, that could potentially be like a small ramp here, which might actually take more energy for me to consume while going through 3 might avoid that ramp. So, now 3 is done. Which is the node I should look at next, which is 6. And So, the neighbor of 6 that I will consider is 5, and because other neighbors are 6 are all done.

So, when you look at the neighbor of 6, so going from through 6 to 5, so 6 already has a distance of 11, which I do not have to go back in the past and examine I already know how much distance I have to travel to get to 6, so it is 11. And I have traveled a further 9 to get to 5. So, the total distance to get to 5 is 20.

(Refer Slide Time: 24:15)

Path Planning

Introduction to Robotics Prof. Balaraman Ravindran

So, 6 is done. And we are actually done completely, because the other node I have to look at this 20 is 4, other node is 5, but 5 is what I am interested in. So, I have reached 5. And there is no way that I can get to 2, I mean, I can get 2 for better than 20 because getting to 5 already needs 20 and vice versa, like I cannot get a cost of 20 better than 20 to reach file because to reach 4 itself, I need 20. Now this is done. And, so what is the best path to get to 5?

Remember that my parent of 5, or the previous of 5 is going to be 6 because that is the one that I used here. So, from 5 I will trace the path back. The previous of 5 is 6, previous of 6 is 3, previous of 3 is 1. So, the path is 1, 3, 6, 5 is the shortest path to reach 5. And likewise, 1, 3, 4 is the shortest path to reach 4, and others are directly connected to 1.

So, have I basically computed the shortest path to reach every one of the nodes here, in the graph. But then I do not have to look at all the nodes in the graph, if I had a specific destination in mind, if 5 was my destination, and suppose there is a whole graph out here, I do not have to look at it, because I have reached my destination. So, I do not have to examine any further, neighbors are 4, primarily because 4 has a cost of 20. Next, I do not have to examine anything further.

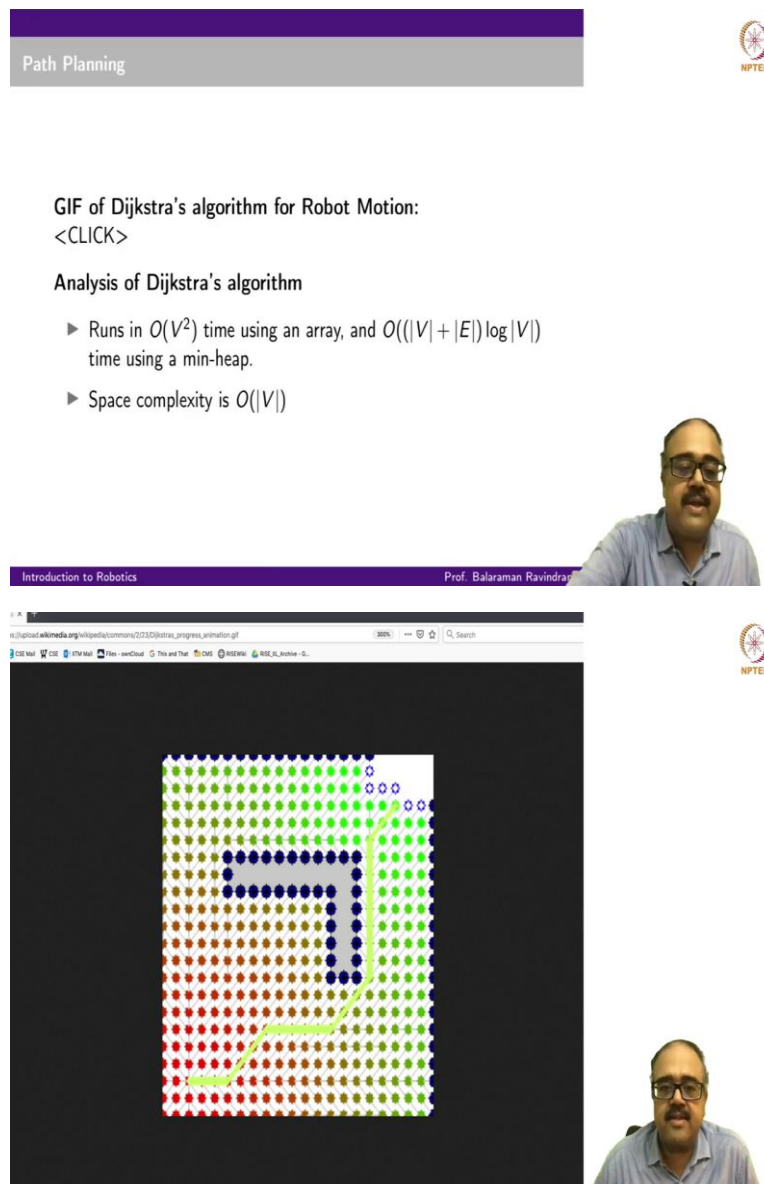
(Refer Slide Time: 25:45)

Path Planning

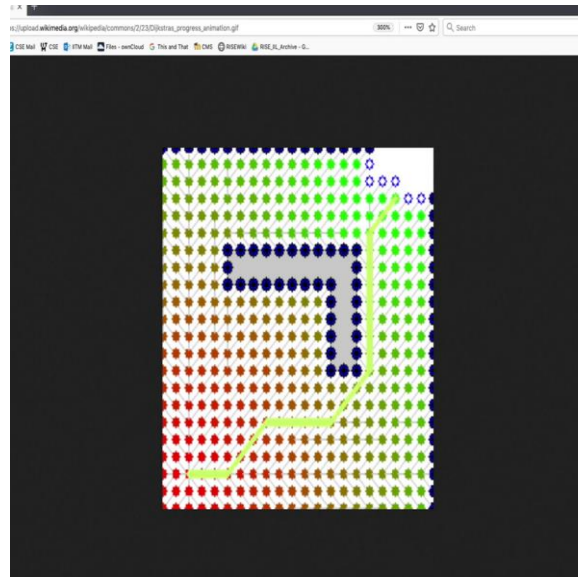
GIF of Dijkstra's algorithm for Robot Motion:
<CLICK>

Analysis of Dijkstra's algorithm

- ▶ Runs in $O(V^2)$ time using an array, and $O((|V| + |E|) \log |V|)$ time using a min-heap.
- ▶ Space complexity is $O(|V|)$



Introduction to Robotics Prof. Balaraman Ravindran



NPTEL

So, let us quickly look at illustration of this. So, I am waiting for the animation to restart, so that we can start from there. So, cut off all this part. So, notice that we started this corner, so then we start examining all these nodes. So, all the red ones are the ones that are done, and the blue ones are the ones that that are yet to be visited and then we continue here. And then when you hit the obstacle, you will see that all of these nodes or do not have any further neighbors to examine, so they are all being stopped.

And then these are all having, the color coding is basically the distance, and so, the redder you are, the closer you are to the goal and greener, you are farther away, and then eventually you will reach the, you will reach the goal. So, this is essentially how the Dijkstra's algorithm is going to work. In fact, here since the cost is all uniform, is very similar to A star search.


But this is how the Dijkstra's, since the costs are all uniform, it is very similar to breadth-first search. But in the case of uniform cost, the Dijkstra's could potentially work very similar to breadth-first search.

So, if you think about how long Dijkstra's algorithm is going to take, if I am using only regular arrays to implement this, it is going to take order of V square, because every time I have to go through it to find out what is the, what is the minimum cost vertex, or minimum distance vertex every time so that I can expand, but I can be more efficient. If we use something called a min-heap, that is always going to return to me the minimum cost vertex with the small cost and the constant cost, in which case I can be more efficient here, so we do not have to really worry about analyzing the algorithm for full efficiency.

And of course, the space complexity is still order of V , the same things that we stored earlier, you have to store now, whether you have visited a node and what is the previous node, so we still have to store that. So, the challenge with Dijkstra's algorithm is that it always looks at how far you have come, so far. From the other hand, if I have knowledge about how far I have to go also, like some kind of an estimate of how much more distance I have to travel, maybe I can be a little bit more efficient in my search.

So, the idea of being able to use this kind of rough estimates, guess, guesses as to how far more I have to go, and so on, so forth, or what are sometimes called as heuristics, is what we use in the next algorithm we look at, which is called A star search.

(Refer Slide Time: 28:25)



Path Planning

A* Search

At each iteration of its main loop, A^* needs to determine which of its paths to extend. It does so based on the cost of the path (stored in g) and an estimate of the cost required to extend the path all the way to the goal (computed by a heuristic function h).

Specifically, A^* selects the path that minimizes:

$$f(n) = g(n) + h(n)$$

The heuristic function is problem-specific. If the heuristic function is admissible - meaning that it never overestimates the actual cost to get to the goal, then A^* is guaranteed to return a least-cost path from start to goal.

Introduction to Robotics Prof. Balaraman Ravindran

So, not only it looks at the cost of the path, so far, which is, which will store in a variable called g , but it also uses, what is called a heuristic to estimate the rest of the cost needed to reach your goal. So, this heuristic will call as h . And so, at any point of time, when I want to select a node, A star selects the node that has the smallest g of n plus h of n . Notice that when you start at the source, you have no, g of n is 0, h of n is the entire distance to the goal, When you are at the goal, h of n is 0, because you have reached the goal g of n is the distance from the source to the goal.

At any point in between, you would have had some distance you have traveled to get there, which is very similar to what the Dijkstra's algorithm has. But then you also have some guess for what is the rest of the distance you need to go. And, A star always uses both, it tries to select a node n , that minimizes g_n plus h_n . In Dijkstra's, what did we do in some sense, we are picking a node that has the least g of n . Here, we look at g of n plus h of n and pick the node that has the least g of n plus h of n , and notice that the heuristic function is problems specific.

There is nothing very generic, but I cannot just say here is a heuristic function that will work all the time, and the heuristic function is also goal specific sometimes, because if I change the goal from the top right corner to the top left corner of the room I might and I have to use a different heuristic, and we would like heuristic functions, which are called admissible. So, admissible is the, the key term here. So, we would like heuristic functions that are admissible.

So, admissible functions are those that never overestimate the actual cost of getting to the goal. It should not say, if it is going to take you say 10 units effort to get to the goal, here heuristic, it is okay, if the heuristic says it takes 9 units, or even okay, if the heuristic says it takes 5 units, that the heuristic should never say it will take 12 units. If the true cost is 10, the heuristic should never say 12, it is okay, if the heuristics says 5. Because that means you will explore it anyway. Because you think it is better path than what it actually is. But if you think it is worse than what it actually is, there is a small possibility that you might never explore it.

And therefore, you might not get a, the least cost path, might not get the optimal path to the goal. So, since you do not want to miss the optimal path to the goal, we assume that here heuristic is admissible.

(Refer Slide Time: 31:27)

Path Planning

```
function A_star(start, goal, h)
// The set of discovered nodes that may need to be (re-)expanded.
// Initially, only the start node is known.
// This is usually implemented as a min-heap or priority queue rather than a hash-set.
openSet := {start}

// For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
// to n currently known.
cameFrom := an empty map

// For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
gScore := map with default value of infinity
gScore[start] := 0

// For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
// how short a path from start to finish can be if it goes through n.
fScore := map with default value of infinity
fScore[start] := h(start)

while openSet is not empty
// This operation can occur in O(1) time if openSet is a min-heap or a priority queue
current := the node in openSet having the lowest fScore[] value
if current = goal
return reconstruct_path(cameFrom, current)

openSet.Remove(current)
for each neighbor of current
// d(current, neighbor) is the weight of the edge from current to neighbor
// tentative_gScore is the distance from start to the neighbor through current
tentative_gScore := gScore[current] + d(current, neighbor)
if tentative_gScore < gScore[neighbor]
// This path to neighbor is better than any previous one, record it!
cameFrom[neighbor] := current
gScore[neighbor] := tentative_gScore
fScore[neighbor] := gScore[neighbor] + h(neighbor)
if neighbor not in openSet
openSet.add(neighbor)

// Open set is empty but goal was never reached
return failure
```

Introduction to Robotics Prof. Balaraman Ravindran



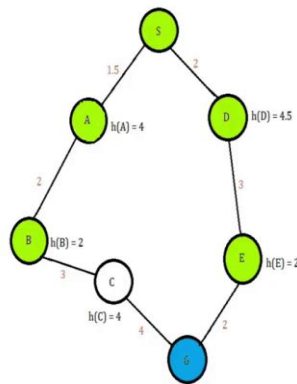
So, let us look at the algorithm. The algorithm looks a little complex, but it is not it is very similar to what he had in the Dijkstra's algorithm case, except that, instead of looking at the g score, which is essentially the distance from the start to the node, I look at the g score plus the h of the neighbor, like the g score of the neighbor plus the h of the neighbor, which is essentially their estimate of the distance to go, the h is initialized a priori which is something that is given to you, the function h is given to you as part of the algorithm specification.

So, all you need to know is the start and the goal, and the whole graph that you are traversing. And the most of the paths are similar, so I start off with saying that all the distance is infinity. And then, I start off by saying all the f score is infinity, and then I say that all the f score has to have. So, the f score of the start state is the h, you remembered, I was telling you that the start is start state basically is all the distance to go. And the goal state is got 0 heuristic, but that comes from the thing.

So, and I also start off with my g score for the start to be 0. Why is that? Because I mean, I am already there at the start state, so there is no g score, remember g is the distance to the node and h is the distance from the node to the goal. And then basically, we just keep updating the, we pick the node that has the least g score plus h score, we pick the node that has at least g score plus h score, and then use that for expanding our search.

(Refer Slide Time: 33:19)

Path Planning



So, let us look at this here how it is going to work. So, here is a simple graph. And I have given the h score for each node here in brackets, so h of A is 4, h of D is 4.5, this is 2, here is 4. And here is 2, notice that this is essentially a overestimate, so if you think of this, so the cost is actually 5. So, the overestimate part we have to, we have to delete. So, we are given the h scores here in brackets, here. So, remember that the h is actually a under estimate, only then it is an admissible heuristic.

And if you think about it, so the distance from D to the goal is actually 5, I think on the graph, and we do not know that yet. If I know what is the shortest path, it is easy, but then my heuristic says it is only 4.5, and likewise, the distance from A to the goal is certainly much larger than 4. But it is okay, because it is still an admissible heuristic, so and then these brown numbers that you see here are actual distances of these edges. These are the weights of the edges. And so, I start off at node, S. And I am going to compute the f score for both A and D.

So, the g score for A is 1.5, and the h is 4, and likewise, the g score for B is 2 and the h is 4.5, so I will take the sum 1.5 plus 4 gives me 5.5 and 2 plus 4.5 gives me 6.5. And therefore, I am going to pick A as the next node to expand. Now, I will expand A. Now, what are the nodes that I have to consider? I have to look at B. What will be my g score for B.? The g score for B is 3.5. So, this is basically the g of A plus the distance from A to B, which is 3.5. And my h score of B is 2. So, the sum of these two is 5.5. So, my f score of B is 3.5 plus 2, which is 5.5. And remember, my f score of D was 2 plus 4.5. Therefore, it is still 6.5. So, that is a bad thing.

Now, what I am going to do is, notice that my admissible heuristic on this path, it is lot lot lesser. So, I am just being more optimistic about the distance from A to the goal from B to the goal, while I am being more realistic, so this is the correct h score, this is the correct distance. And this is more or less correct and putting 5 to 4.5. So, the guys were more optimistic. I am exploring them first. But I eventually get to D as well. So, what I am going to do next is explore B.

Now the two options that they have to consider are C and D. So, what is the g score for C is the g of B, which is 3.5, plus 3, so 6.5 is the g score for C. And the h score for C is 4. So, I because I am next to the goal, so I can always initialize it to the correct distance. Therefore, my f of C is 6.5, which is the g score plus 4, which is the h score gives me a total of 10.5. And my D was 6.5. So, I am going to finally expand my D. So, I am not expanding my C, I am going to finally expand my D, and therefore what do I get, my new competitors are E and C. C we already know is 10.5.

What about E? E is the g score of D plus the distance which is 3, so 2 plus 3, which is 5, so g score of E is 5. And the h score of E is 2. So, the f score is 5 plus 2, which is 7. And, therefore I am going to expand E next, 7 versus 10.5. I am going to expand E next. And, so, E's g score is 5, and the distance to G is 7, of course, the h score of G is 0. So, I get 7 and I have reached G, so now I will backtrace the parent node, remember, we had the parent. So, the parent for G was E, the parent for E was the parent for D was S.

And therefore, that is my shortest path. It is a very simple example, so that we can see what is happening, and notice that we never expand at C. If I had done breadth-first, I would actually expand at C also before getting to G. But I never expand C. And also, the problem here is because my heuristic function was a little off, If it had been a little bit more clever about this heuristic, this heuristic of being for let us say, that means 7, which is more likely, already is even 7 is an underestimate, because the distance from A to G is 9.

So, even if my heuristic had been 7, I would have gone down this path, because this path, my total distance itself is 7. And here, my h score, my f score would have been 8.5. So, I would not have expanded this whole path, because my heuristic was a little off, I ended up doing the, this path a lot, before I got down to the correct path to get to the goal.

(Refer Slide Time: 38:53)

Path Planning

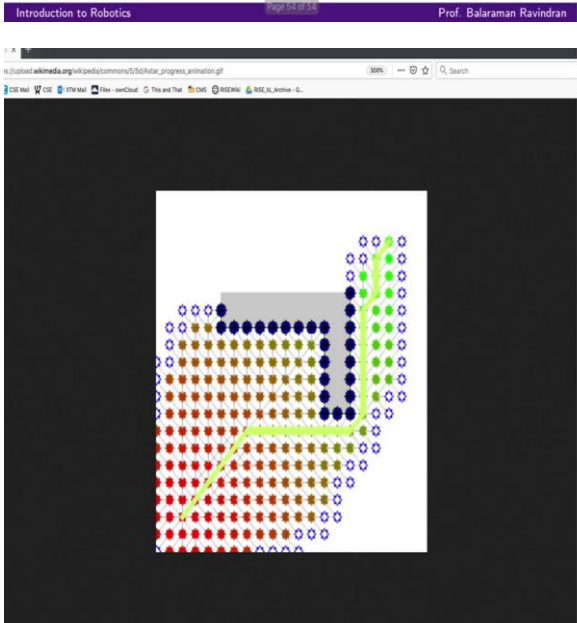
GIF of A* Search for Robot Motion:
($h(x)$ used here is the straight-line distance to the target point)
<CLICK>

Analysis of A* Search

- ▶ The time complexity of A* depends on the heuristic.
- ▶ Assuming that the goal exists and is reachable from the start state, in the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the shortest path d : $O(b^d)$, where b is the branching factor (the average number of successors per state).

The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes that an un-informed search would expand.

Introduction to Robotics Prof. Balaraman Ravindran



So, we will see how this works in this particular example. Again, waiting for the start this time. So, here is the heuristic, I am using this the direct distance to the goal, like just draw a diagonal line, whatever is the shortest path to the goal, assuming there is no obstacle, you can see that when I started here, it directly expanded that line because this is the shortest path to the goal. And my heuristic says that, hey, this is going to take you to the goal very quickly. Therefore, I only expanded that path, and now after that, I have just expand this whole region behind obstacle.

But once I got to a point, notice here now I am expanding the whole region that is blocked by the obstacle because it is still trying to go to the shortest path to the goal. Then once I reach a point where I get a direct line of sight to the goal, there is no obstacle here anymore. It will

now only go through this path, that path stops and only this is expanded and then I finally get to the goal. And, this is the path to the goal.

So, a large fraction, if you remember, in the Dijkstra's, we had expanded almost up till here, in the extras, whereas we had expanded a region almost up till here. Before, we found the path shortest path to the goal. And in A star, we are finding it while a large number of nodes are still not expanded. So, it turns out that this is actually a reasonable heuristic to have. And quite often, that is what the heuristic that we will apply in the path planning problems. It is essentially the distance as the crow flies to the obstacle and to the, I mean to the target. And that will always be admissible heuristic.

And the time complexity of A star really depends on the heuristic, how hard is the heuristic to compute. And, the point is, if b is the branching factor, branching factor is the number of neighbors that we typically on an average have. So, in our case, the number of neighbors was 8, so the branching factor was 8. And the shortest path to the goal, let us say is 10 steps, If I am doing dumb breadth-first search, I will really be expanding something of the order of 8 power 10, the huge number of nodes.

But then if I am using A star, I can prune away a large fraction of those, because they are too far away on the from, by as measured by the heuristic, so anything that is going to take me farther away from the goal, I will ignore. And therefore, something like A star where the heuristic allows me to, kind of focus my search is a much more efficient way of finding paths.

In fact, if you in practical algorithms, practical implementations A star and for most simple robot navigation problems, so these were the, these are the kind of the workhorses like BFS, A star and Dijkstra's are the workhorse algorithms for path planning. But in more complex environments, people prefer what are called probabilistic path planning algorithms, where you do small amounts of sample of the world to figure out whether you are going the direction, and these return feasible paths and might be optimal, but they do not typically, do not guarantee optimality they return feasible paths, but they do so much more efficiently than something like A star or Dijkstra's.

So, we have looked at multiple topics here, but we have covered it, covered these topics that are very very high level. So, starting from the notion of state representations, and state estimation, (())(42:44) filtered with different variants of base filters, looking at measurement

models, motion models, map building, localization and path planning. And we have again touched upon these topics just to give you a flavor of what are the various things that go into building good robotic systems, and this in conjunction with whatever else you have seen already in the course, should be should be enough to get you started, but no means sufficient for you to say that I know how to do robotics not.

And move largely the way to way forward is to get your hands dirty. I mean, start building small robot models and try to play around with those. Thank you.