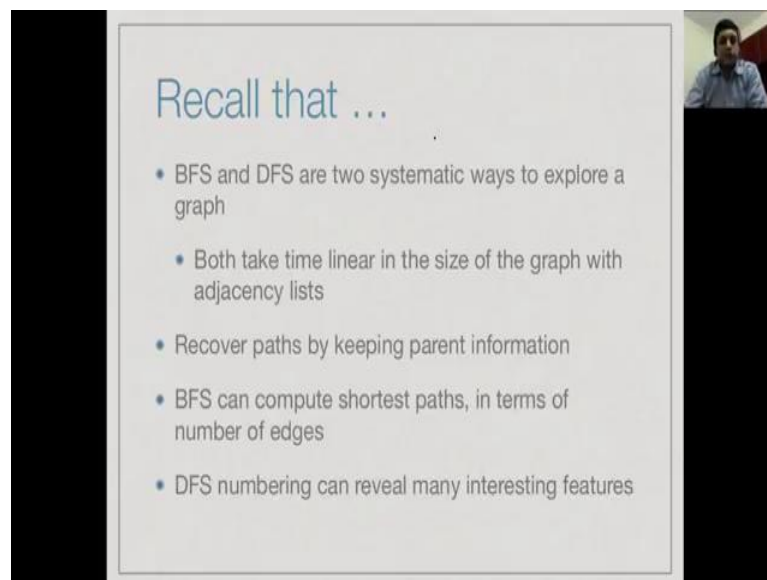


Module – 01
Lecture - 25
Shortest Paths in Weighted Graphs

Let us turn our attention now to weighted graphs, graphs in which edges have cost associate with them. The most important thing that we can do with weighted graphs is to compute shortest paths.

(Refer Slide Time: 00:14)



Recall that ...

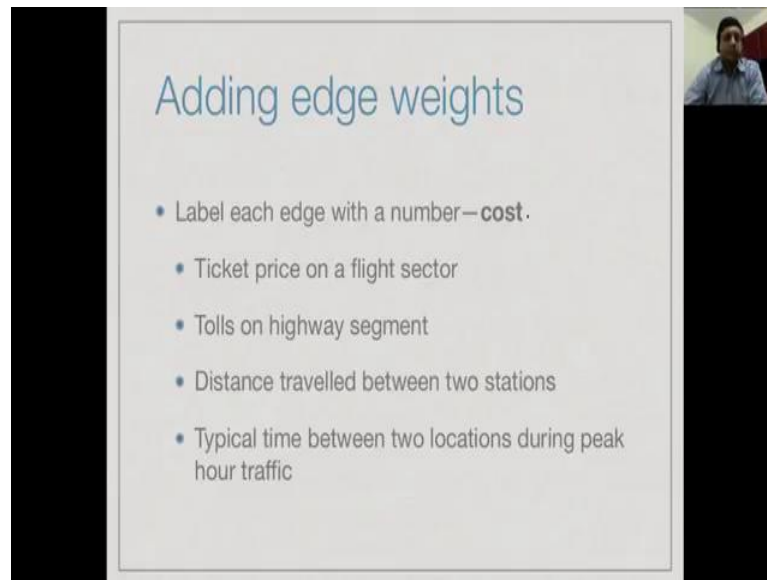
- BFS and DFS are two systematic ways to explore a graph
 - Both take time linear in the size of the graph with adjacency lists
- Recover paths by keeping parent information
- BFS can compute shortest paths, in terms of number of edges
- DFS numbering can reveal many interesting features

So, let us review what we have seen so far. So, we have been looking at unweighted graphs that is sets of vertices and sets of edges, where each edges is just the connection between two vertices. We have seen two systematic strategies to explore such graphs, breadth first search and depth first search. Now, both of these will be linear in the size of the graph, if we use adjacency lists representation for the edges.

Now, we saw that while we explore a graph using either BFS or DFS from a given vertex, not only do we discover which vertices are connected to this start vertex. We can also recover the path back to the start vertex by keeping extra information, such as the parent of each node, we visit. We have observed that breadth first search, because if it is layer by layer strategy, actually uncovers the shortest distance to every vertex in terms of the number of edges from the start vertex.

And depth first search, though it does not find the shortest path, it gives us a lot of other information in terms of the order in which we visited. So, by keeping track of the pre and post DFS numbering, we can actual recover structure of properties of the graph.

(Refer Slide Time: 01:22)

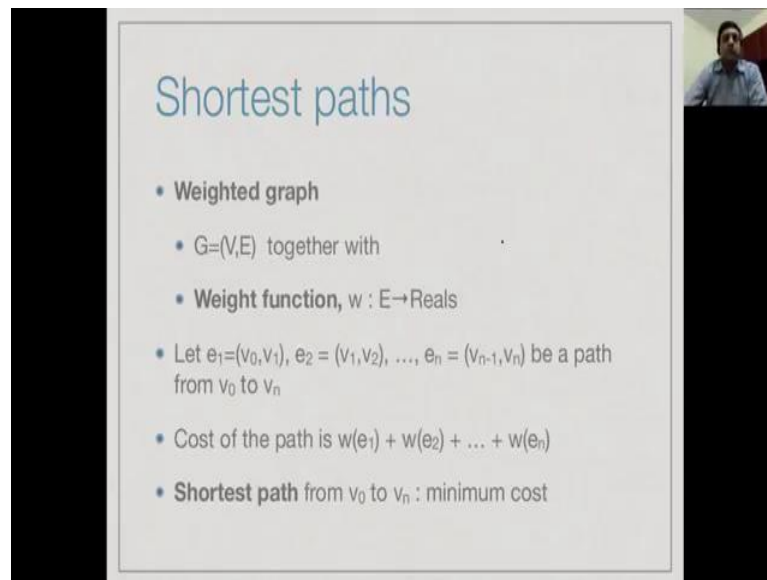


Adding edge weights

- Label each edge with a number—**cost**.
 - Ticket price on a flight sector
 - Tolls on highway segment
 - Distance travelled between two stations
 - Typical time between two locations during peak hour traffic

Now, we look at what happens if we add costs to the edges. Now, depending on the application, this cost of several natural interpretation, for example, in the initial example of a graph, we look at an airline routing map. So, on such a graph, the edge weight, the cost associated with an edge could be the ticket prize on the flight or if edges represent roads in a network of highways, then we might have a cost representing the toll that one has to pay on a particular segment of road. Or, if you have a railway network, it might represent the distance between two stations or on a simple traffic road map of a city, it might represent how much time one might except to take between two intersections at busy times.

(Refer Slide Time: 02:12)



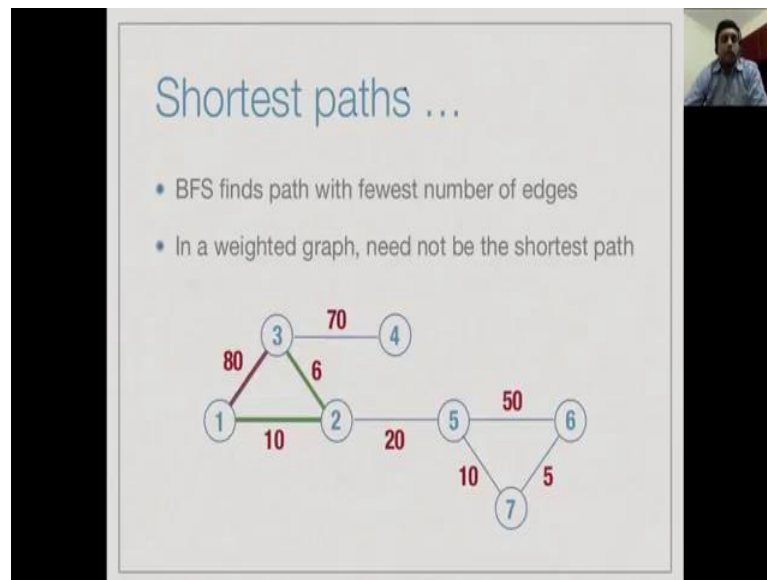
The slide is titled "Shortest paths" in a blue font. It contains a bulleted list of definitions and concepts. In the top right corner, there is a small video inset showing a man in a light blue shirt speaking.

- **Weighted graph**
 - $G=(V,E)$ together with
 - **Weight function**, $w : E \rightarrow \text{Reals}$
- Let $e_1=(v_0,v_1)$, $e_2=(v_1,v_2)$, ..., $e_n=(v_{n-1},v_n)$ be a path from v_0 to v_n
- Cost of the path is $w(e_1) + w(e_2) + \dots + w(e_n)$
- **Shortest path** from v_0 to v_n : minimum cost

So, in general, weighted graph is just a normal graph along with an extra function which tells us the cost of each edge, is usual to call this a weight function. So, weight function just assigns each edge in E some number, so these think of some real number. So, now, if you have a path from v_0 to v_n , so we have a path from v_0 to v_n , we have a sequence of n edges, $v_0, v_1, v_1 v_2$, each of these in our weighted graph would have a weight and a natural thing to do would be to add the cost of these weights.

So, supposing these are distances, then the total distance should be followed this path and we want to be a, will be a sum of the distances. If it is a sequence of flights, the total cost that we pay for a ticket will be the sum of the cost. So, we will extend weights from edges to paths by just adding up the sum of the weights along each path. And now, a natural problem that we want to solve is to find the shortest path between a given pair of matrices; that is what is the minimum cost that I can incur to go from v_0 to v_n .

(Refer Slide Time: 03:19)



So, we have seen before that breadth first search will solve this problem, if our cost is in terms of number of edges. Another way of saying it is that we can assume that each edges has a same cost, 1 or any fixed number. So, that as I traverse more edges, the cost is propotional to number of edges for us. But, if you do not have this property, the cost can be arbitrary, then breadth first search does not work.

For instance in this example, we see that between 1 and 3, there is a direct edge, but it is weight is 80 and we can do a shorter cost path from 1, 2, 3 by going by add 2. So, it has two edges, the total cost that you need 10 plus 6, which is 16. So, the shortest path in a weighted graph need not be the shortest in terms of number of edges, we are talking about the sum of the cost along the edges. So, we might have a longer path in terms of number of edges, but shorter total cost. And our goal is to compute such paths regardless of the actually length of the shortest path.

(Refer Slide Time: 04:18)



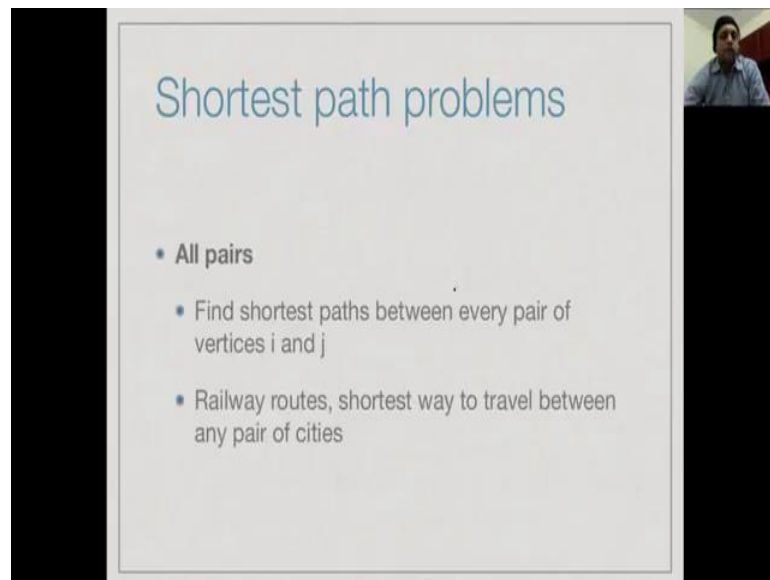
Shortest path problems

- **Single source**
 - Find shortest paths from some fixed vertex, say 1, to every other vertex
 - Transport finished product from factory (single source) to all retail outlets
 - Courier company delivers items from distribution centre (single source) to addressees

So, we can ask two types of question regarding shortest paths, one is so called single source path. In the single source shortest path problem, we have a designative place from where we start all our paths and we want to find the shortest distance to every other path. Now, this (Refer time: 04:37) has very natural application, so supposing you are a manufacturer and you have a factory where you make all your items, now you have to distribute these items across the country.

So, how do you find the shortest way to transport your items in the factory which is a single point to all these places, where it can be sold or all the retail outlets or you might be a courier company. Now, courier company will shift all items to a given street, it was a centralized office and from this centralized office, it has to be distributed to all parts of the city. So, how do you compute the shortest path from the distribution office to all the different addressees to which the courier items have to be delivered?

(Refer Slide Time: 05:15)



Shortest path problems

- All pairs
 - Find shortest paths between every pair of vertices i and j
 - Railway routes, shortest way to travel between any pair of cities

And other problem would be to compute the shortest distance between any pair of cases or any pair of vertices. So, this would be a natural problem for example, if you have a routing network of an airline or a train network. Now, whenever you want to travel from some city a to city b , you would like to compute the shortest path between a and b or instance, if you use service like Google maps and you say that you want to go from a source to a destination, it will try to compute the shortest path and give you the time in terms of walking or driving, etcetera.

So, we have two different types of problems, one we may have a single source and we want the shortest path to every destination between from that source and other version where that we want to find the shortest path between every pair of vertices.

(Refer Slide Time: 06:00)

This lecture...

- Single source shortest paths
- For instance, shortest paths from 1 to 2,3,...,7

```
graph LR; 1 ---|80| 3; 1 ---|10| 2; 3 ---|6| 2; 3 ---|70| 4; 2 ---|20| 5; 5 ---|50| 6; 5 ---|10| 7; 6 ---|5| 7;
```

So, we will begin by looking at this single source shortest path problem. So, we have to design, we have to designate in such a problem, what is the start vertex. So, maybe our start vertex that is we can mention assume, it is the vertex 1. Remember that we usually call our vertices 1, 2, 3 up to n. So, here we have a graph with 7 vertices and the edge weights are written by the side of the edges into red and we want to start at 1 and find out, how to get from 1 to every other vertex in the shortest possible cost and the systematic way to compute this.

(Refer Slide Time: 06:32)

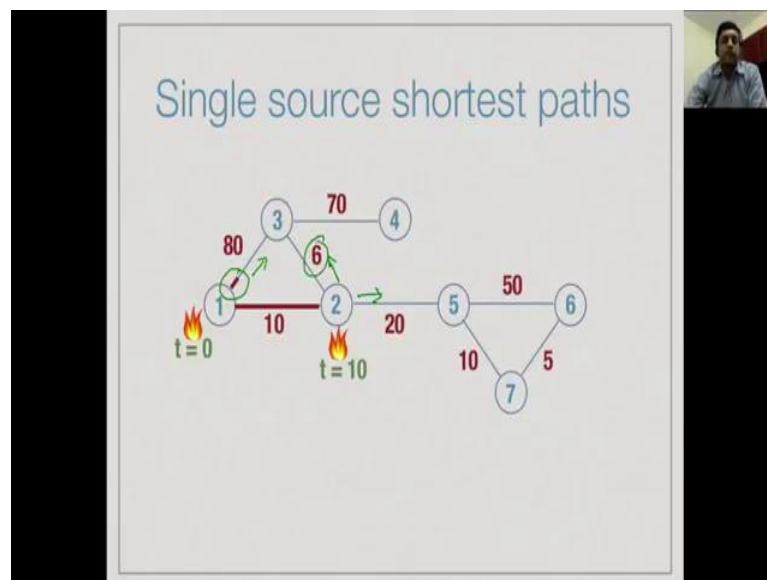
Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
 - Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 1 is nearest vertex
- Next oil depot is second nearest vertex
- ...

So, here is an analogy which will help explain the algorithm. So, let us assume that every vertex is an oil depot and all the edges are pipelines and the pipelines are the cost of the lengths of the pipeline. Now, when we set fire to this original vertex, start vertex 1 and fire will catch on all the pipes connected to vertex 1. Assuming that the pipe the fire travels at uniform speed, it will reach the shortest, the closest vertex first.

So, the first vertex first oil depot that catches fire after the first vertex is a nearest vertex to 1. Then, the second oil depot that catches fire is a second nearest vertex path and assuming this fire is traveling at a uniform speed by measuring the speed at which the fire reaches each vertex, we will actually end up with quieting the shortest path.

(Refer Slide Time: 07:31)

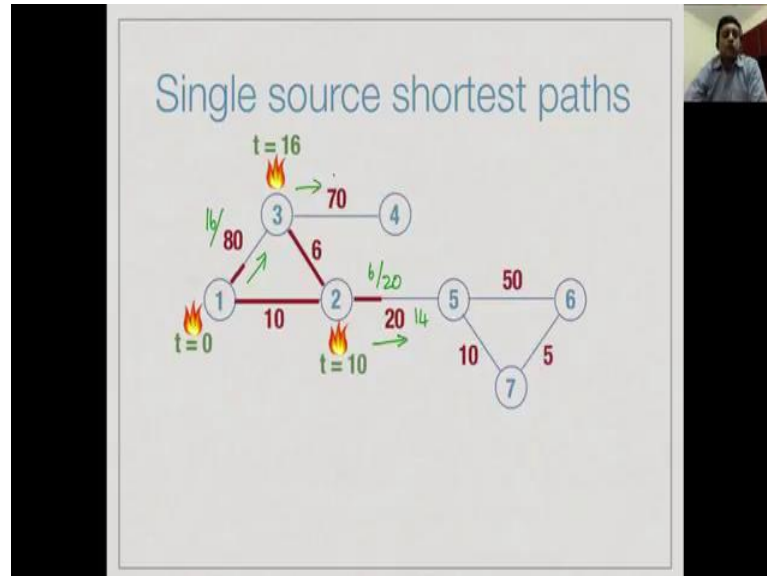


So, let us try to execute this analogy, this strategy on this particular, so we begin a setting fire to this source vertex at times 0 and now a fire will start going along the edge 1, 3 and the edge 1, 2. Now, since the edge 1, 2 is shorter, in 10 units of time vertex 2 will burn. At this point, we have indicated by the fact that there is a small burn portion here, that there is a partial fire going from 1 to 3, but it is only travel 10 out of 80 units, only one eighth of the way it is travelled from 1 to 3.

So, at t equal to 10 vertex 2 burns and we can say that this is the shortest cost of the shortest distance to vertex 2. Now, the fire continuous to propagate through 2 in two directions, from 2 to 3 and from 2 to 5 and we will while of course, the fire from 1 to 3

also continuous in the same rate as it was before. Now, we can see that after 6 units of time, vertex 3 will burn.

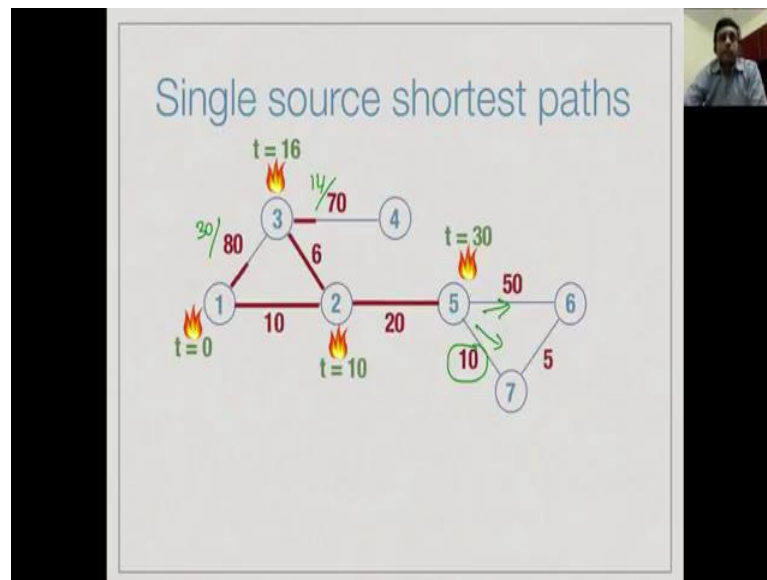
(Refer Slide Time: 08:34)



So, at $t=16$ vertex 3 has burnt, now this has the fire from 1 to 3 has travelled to 16 by 80 of it is distance, but the fire from 2 to 5 has travelled 6 by 20 of this means. Now, haven't burnt 3 and new fire will continue in this direction, this old fire from 1 to 3 continuous as before and of course, the fire 2 to 5 means as before. So, now, we have to see, which of this will reach its destination first.

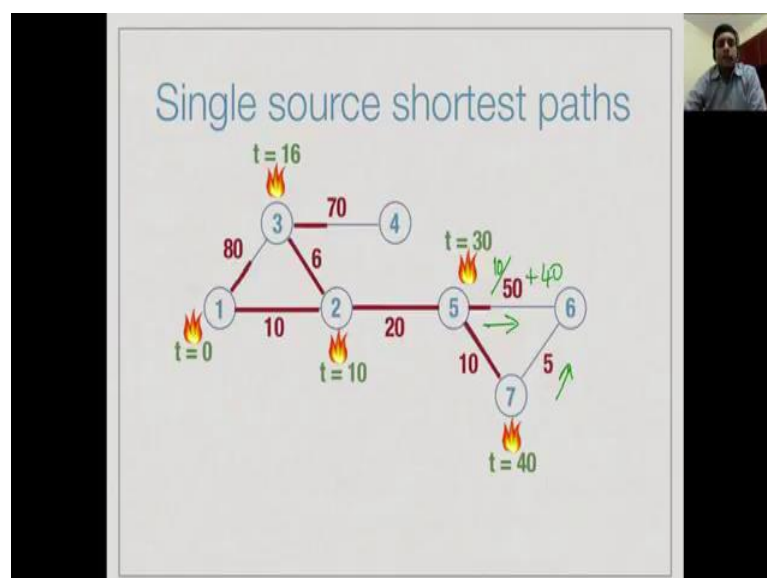
So, we can see that the fire from 2 to 5 has in 14 units of time; you will reach 5 by which time neither it is 3 nor 4 route of them. So, if you continue other 14 and other 14 units of time by t equal to 30, we find that vertex 5 burns.

(Refer Slide Time: 09:15)



And here we have done 14 out of 70 from 3 to 4 and we have done 30 out of 80 totally from 1 to 3. So, these fires are still on their way from the source, the starting point of the edge to the ending point of the edge. Now, continuing in this way, the fire now propagates from 5 to 6 and 7 and we can see that in 10 units of time, we will find the vertex 7 burns.

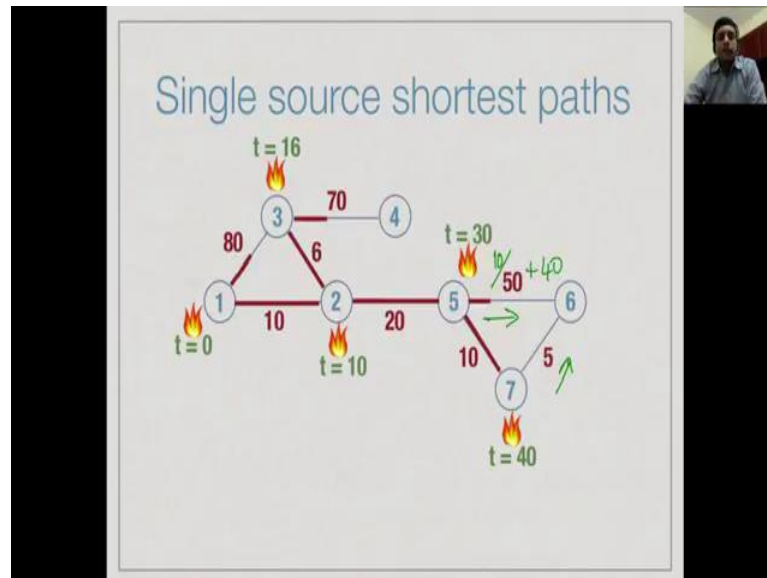
(Refer Slide Time: 09:42)



And now, from 7 a new fire starts towards 6 is an old fire coming from 5 to 6, but this will overtake it, because this is only done 10 by 50. So, there is still 40 units of time to

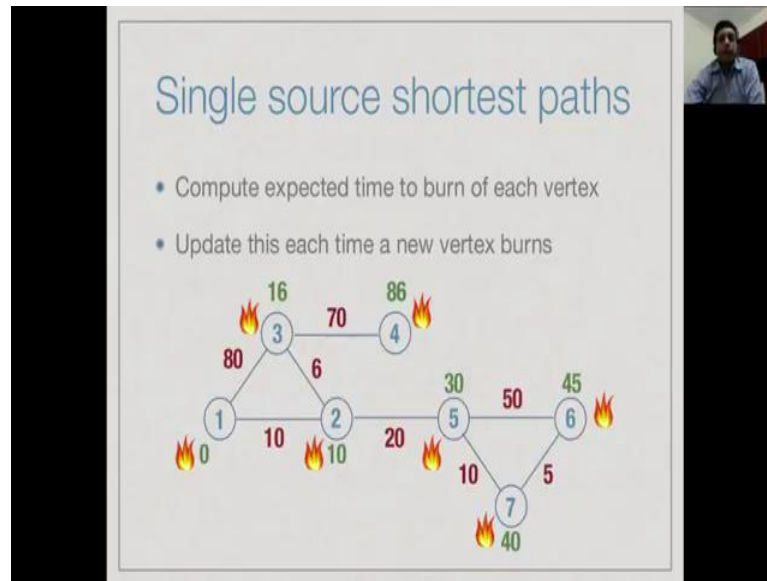
go before the fire reaches 6 from 5, but in 5 units of time, it will be reach from 7, so therefore, at t equal to 45, you find that vertex 6 burns.

(Refer Slide Time: 10:02)



And now the only vertices, which are (Refer time: 10:08) to burn, are actually 3 and 4. So, at this point we continue, we will find that at t equal to 86, actually sorry, vertex 3 is already burnt to that fire did not reach vertex 3. But, only your takes remaining to burns was there are 4 and at time it is 86, that is 70 units after the fire after the fire started at the begging of the edge, it finally reaches and now everything is burnt. Now, we have label each vertex by the shortest time it took from for the initial fire to reach them and this, we claim is the shortest path to reach vertex from that start vertex path.

(Refer Slide Time: 10:50)



So, let us see, how we would actually compute this burning process that we describe. So, what we do is we associate time to the shortest distance as a quantity with each vertex and initially, we do know anything. So, we assume that the vertex is not reachable, that is a shortest cost of every vertex is infinity. Now, we know that the vertex 0, vertex 1, the start vertex is reachable in no time at all.

So, we assign its cost to be 0 and having assigned it is cost to be 0, we can now recomputed, it is neighbors cost as the minimum of the cost that you can reach through 1 and there will cost at we already (Refer time: 11:24). So, from 1 to 3, we can go and time 80 we know, we believe right now is infinity. So, we know that we can improve the infinity to 80.

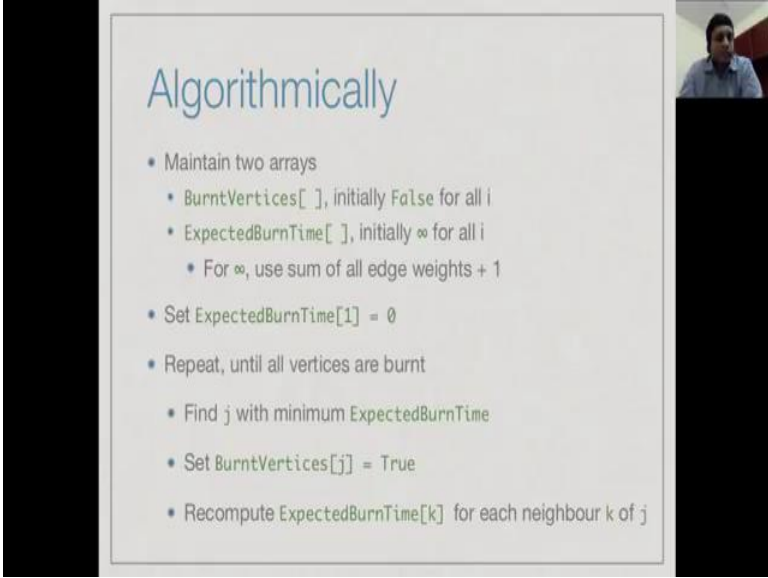
Similarly we can improve the infinity of 2 to 10, so in this process, we get two updates, now all the other vertices, we still do not know, how to reach them, so they would not get changed. Now, among these 2, if find that 10 is smaller than 80, so 10 now they can be no short up part to them, so we say the 10 is burnt and we update it is neighbor again. So, we say that vertex 2 burns at time 10. Now, because it burns at time 10, we can update the cost to 3 as the minimum of 10 plus 6, 16 or 18, which is 16 and now, we can reset 5 from infinity down to 10 plus 20 is 30.

So, we continue in this way. Now using the fact that now among these things that are going to burn, 3 is going to burn in 16 and 5 is going to burn 30, it is clear that 3 burn

next. So, we reset its status to burn and we have take it is neighbor namely 4 to 16 plus 70 is 86. Now, among those which are not burnt, namely 4 and 5, which have a finite value, 5 will burn next.

So, we set its status burn and we updates it is neighbors to 80 and 40, at 6, we have 80 and at 7, we have 40, now 7 burns next. So, now, we look at the cost going out of 7 and we replace the 86 by 45. Now, 45 burns next, 6 burns next at 45, so we mark it burned and finally, you mark 4 burn at 86.

(Refer Slide Time: 13:00)



Algorithmically

- Maintain two arrays
 - `BurntVertices[]`, initially False for all i
 - `ExpectedBurnTime[]`, initially ∞ for all i
 - For ∞ , use sum of all edge weights + 1
- Set `ExpectedBurnTime[1] = 0`
- Repeat, until all vertices are burnt
 - Find j with minimum `ExpectedBurnTime`
 - Set `BurntVertices[j] = True`
 - Recompute `ExpectedBurnTime[k]` for each neighbour k of j

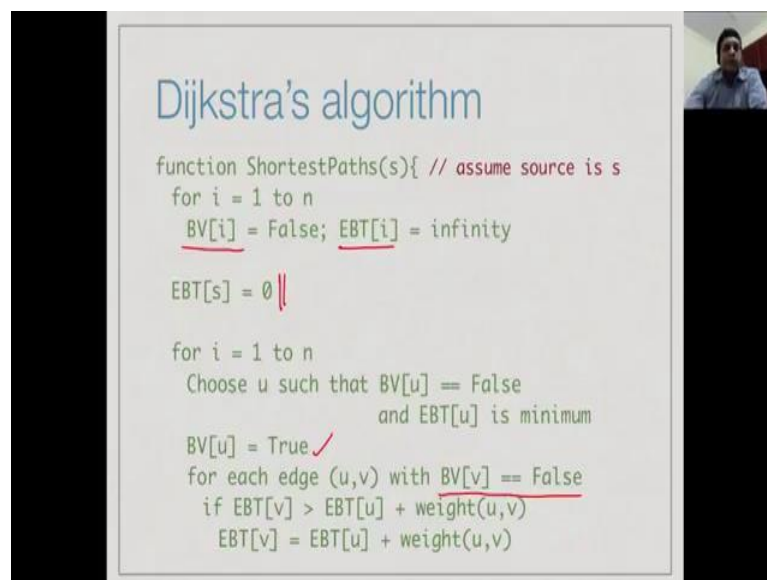
So, to write this formally as an algorithm, we maintain this information about the burnt, but vertices and the time, it is take to burn a vertex in two different arrays. We have an array called burnt vertices which is Boolean vertex a Boolean array. So, it tells us whether a vertex has been burnt vertices and then, we have a expected burn time array, which tells us the time at which vertex is believe to burn.

So, initially, we said that no vertex is burn, so we said all the vertices to have burnt vertex value false and initially, we said the expected time for every vertex to infinity. Now, infinity of course, is a concept which is difficult to estimate, but we can easily check that no vertex can be at a distance, which is bigger than the sum of all the cost in the graph plus 1.

Because, at most you can actually the shortest path that actually take even fewer edges, but at most, you can use every edges, there is no point in using in edge twice, at most you can use every edge once. So, if you add all the cost in the graph and add one to it and then, this can definitely being treated as a infinity, no actual cost can be larger than this cost.

So, we can use infinity, but they infinity, there is a concrete strategy, it was assign with a infinity. Now, we start with the start source vertex which we assume is 1 said at 6 is burn time to 1. And now, we repeatedly look at the smallest expected burn time, burn it next and reset the expected burn time for all it is neighbors, based on the current burn time of that neighbor and the minimum of that and the value of burn time, through this vertex.

(Refer Slide Time: 14:40)



Dijkstra's algorithm

```

function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    BV[i] = False; EBT[i] = infinity

  EBT[s] = 0 ||

  for i = 1 to n
    Choose u such that BV[u] == False
                        and EBT[u] is minimum

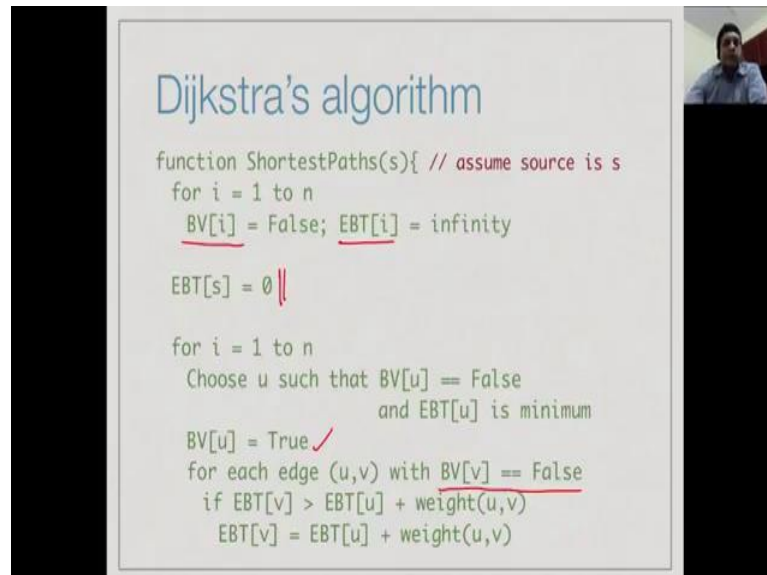
    BV[u] = True ✓
    for each edge (u,v) with BV[v] == False
      if EBT[v] > EBT[u] + weight(u,v)
        EBT[v] = EBT[u] + weight(u,v)
  
```

So, this is very simple algorithm, so here it is in Pseudo code. So, we have these two vertices burnt vertex array and we have the expected burn time arrays. So, we initialize burnt vertex to false and expected burn time to infinity for vertices. Now, we said the source vertex expected burn time to 0 and now, we choose the smallest unplanned vertex.

So, we look for u, which has not be burned and how expect burn time is minimum, make it burn and for each of it is out going neighbors, which is not burnt. If the current estimate for that is bigger than the estimate to going through u, that is the burn time of u plus the way from u to v. Then, you update the burn time to be the new weight, so in

actual algorithm burn time and expected burn time have a natural interpretation. So, burn is just visited.

(Refer Slide Time: 15:35)



Dijkstra's algorithm

```
function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    BV[i] = False; EBT[i] = infinity

  EBT[s] = 0

  for i = 1 to n
    Choose u such that BV[u] == False
                        and EBT[u] is minimum
    BV[u] = True ✓
    for each edge (u,v) with BV[v] == False
      if EBT[v] > EBT[u] + weight(u,v)
        EBT[v] = EBT[u] + weight(u,v)
```

So, when we are visitors about vertex, so we have burn it and the distance is the just the expected one time. So, we can just rewrite exacted these the same algorithm, it just the B V has been replace by have visited and EBT has been is required by distance. So, we said visited to false distance to infinity, start with the initial distance to this the start vertex is 0.

And then, it will frequently for each unvisited vertex, so these minimum distance, set it be the visited and updates itself. This is the Dijkstra's algorithm for single source shortest paths.