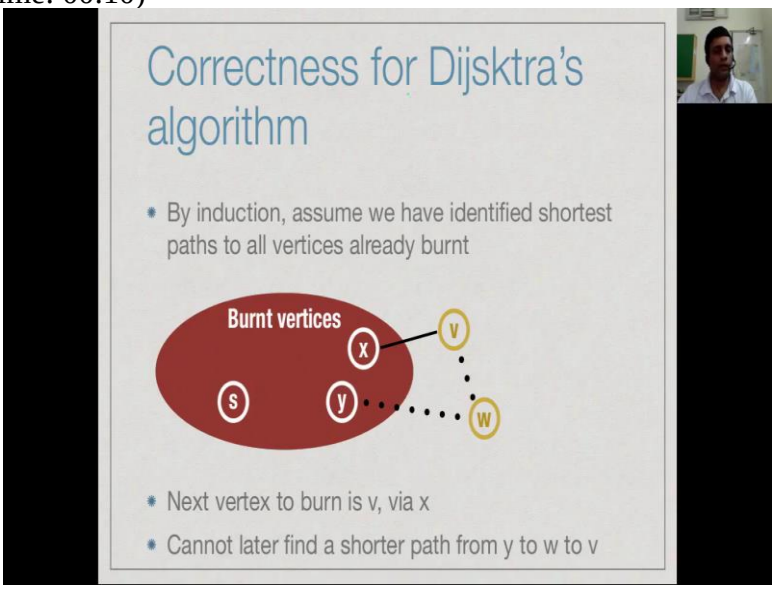
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Module – 03
Lecture - 27
Negative Edges: Bellman-Ford Algorithm

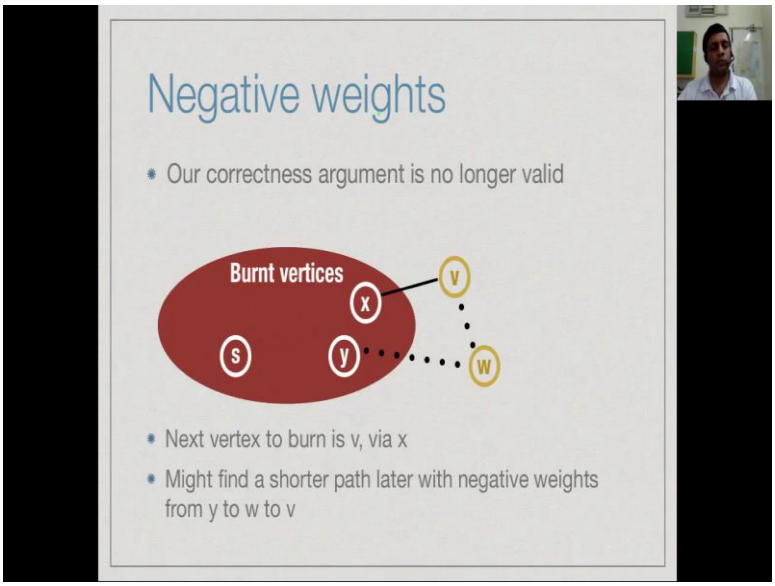
Now, let us look at shortest paths in graphs where we allow Negative Edge weights. In particular let us look at the Bellman-Ford Algorithm.

(Refer Slide Time: 00:10)



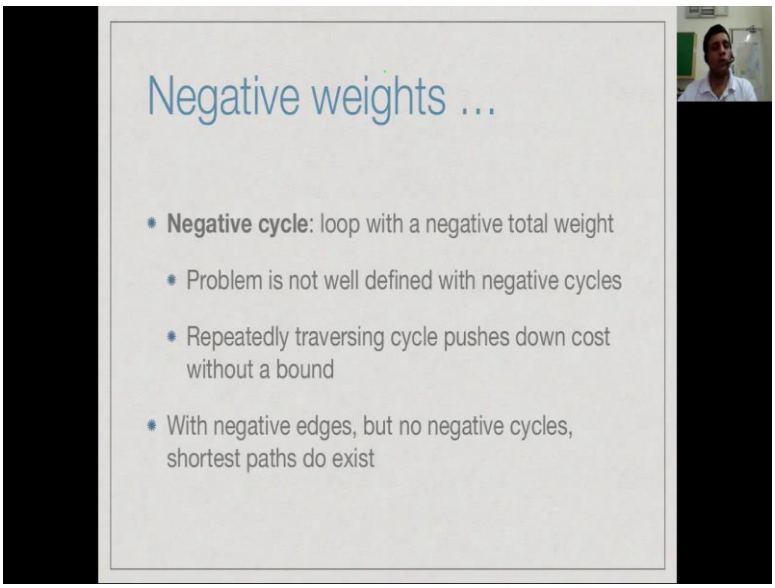
So, recall that the correctness for Dijsktra's algorithm relied on an invariant property that every vertex that we burn automatically has the shortest path computed at the time when we burn it.

(Refer Slide Time: 00:25)



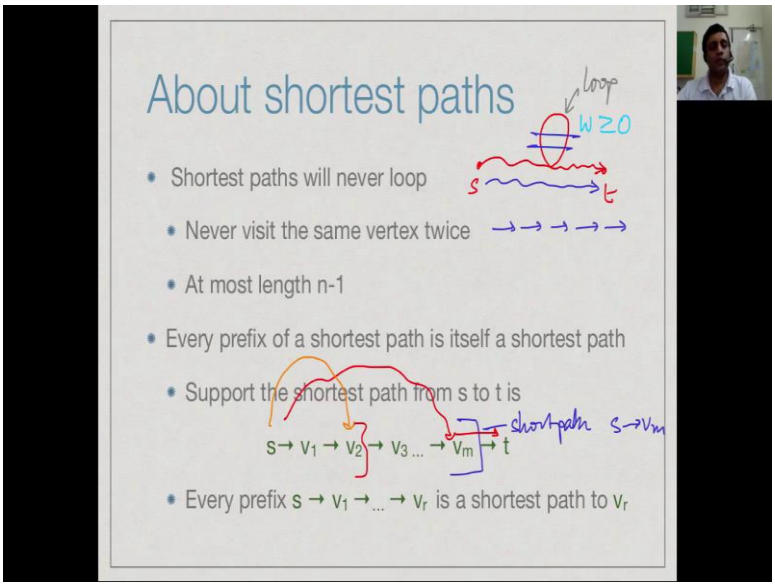
However, as we saw this argument does not work if we can have negative edge weights, because we may find later a path via vertex which we have not burnt yet which becomes a shorter path to a vertex we have burnt earlier.

(Refer Slide Time: 00:38)



We also said that allowing negative edge weights is one thing, but allowing negative cycles is not a good idea. Because, once you have a negative cycle, you can go around, around cycle as many times as you want, it arbitrarily reduces the length of the path, so the shortest path is not even a well defined quantity. So, long as we have negative edges, but not negative cycles, shortest paths do exits and we can hope to compute them.

(Refer Slide Time: 01:04)



So, let us first look at two basic properties about shortest paths which hold regardless of whether the edges can be negative or not, provided we do not have negative cycles. So, the first property is fairly obvious, that is a shortest path will never go through a loop. So, supposing I want to go from s to t and suppose along the way I actually go through the same vertex twice and then I continue.

Now, what we know is that this is a loop and since it is a loop, the weight is greater than or equal to 0, it cannot be negative, because we have ruled out negative loops. So, therefore, if I take this path and I just cutout this loop, then I have a direct path from s to t. And if I look at the cost of direct path from s to t, it cannot be any bigger than this one, because at best this loop has 0 and decrease nothing but, in the general case the loop has some positive cost and actually reduce the cost.

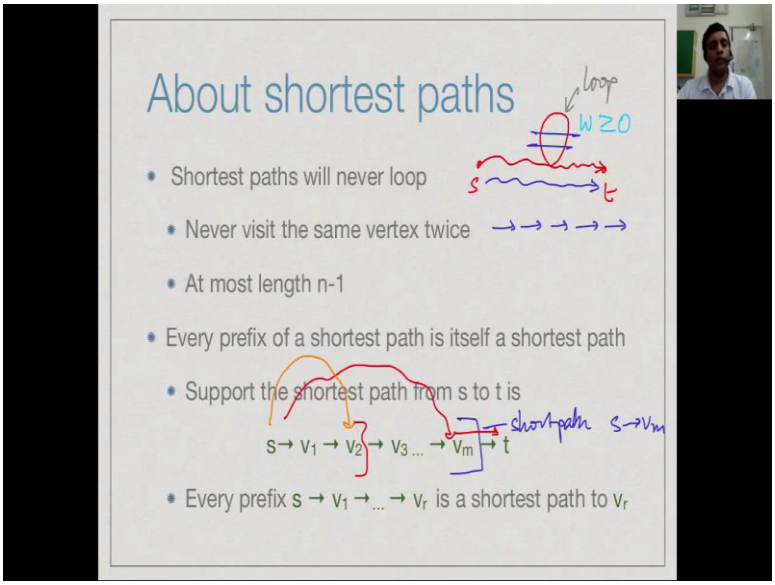
So, shortest path never loops, so it will never does not sent vertex twice, this means that I at most have at most n minus 1 edges. Because I can only allows, so I totally have n vertices, so if I do not allow any vertex to repeat anywhere along the way then; obviously, I can only have n minus 1 vertices, n minus 1 edges. So, there is a bound on the length of the shortest path, the other property is that regardless of how the shortest path looks, along the way every path that makes up the shortest path is itself the shortest path.

So, for instance I have a path like this from s to t which goes through some intermediate vertices v 1, v 2 up to v m. Then, if I look at the path only up to v m, then this is our

shortest path from s to v m. There cannot be any shorter path, supposing there was another shorted path. Suppose, the shorter way to get from s to v m, then I can take this and this and the red path now will be shorter than the green path.

So, therefore, the fact that I am going to t via v m and this is a shortest path means that there must also we only this much of our short path from s to v m and this whole that we have point. So, again what happen s to v 2 this must be the shortest path, because otherwise if there were a shortest path then I will take that shorter path and then I will add on this path which I already have and I will get a shortest path to everything from v 3 onwards. So, every prefix of a shortest path is itself a shortest path. So, these two properties are enough for us to arrive at the Bellman-Ford algorithm for shortest path in the presence of negative edge weights.

(Refer Slide Time: 03:36)



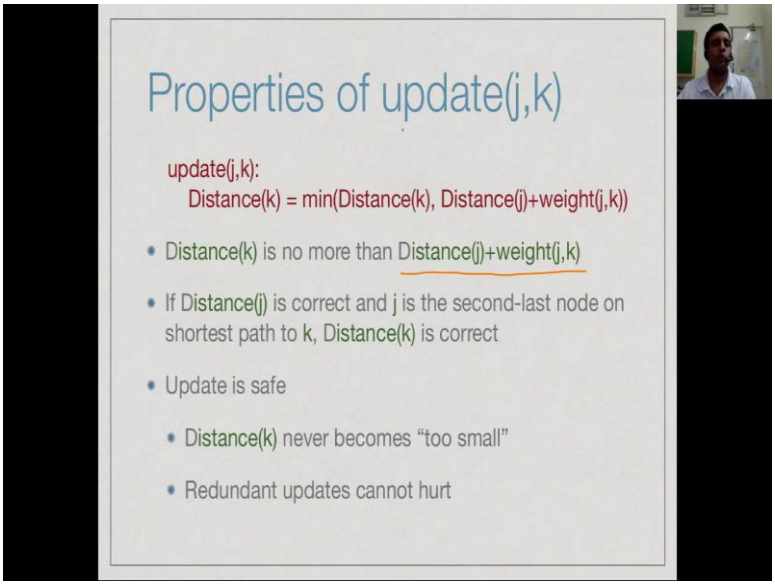
So, to get to the Bellman-Ford algorithm we have to analyze a little bit more about what is happening in Dijsktra's algorithm. So, in Dijsktra's algorithm whenever we burn a vertices, whenever we visited, we updates all it is neighbors. So, we update the distance of a when we burn a vertices j, we update for every edge j k, we update the distance to k to be what distance we already know for k together with a distance to j and h from j to k.

So, we have a newly discovered distance through j perhaps and we compare it to the distance we already know and we keep the smaller of the Dijsktra's algorithm. So, let us call this operation an update, it updating k from j. So, in Dijsktra's algorithm we only do this update when we burn j and the correctness of Dijsktra's algorithm says that when we

burnt j, the distance to j is the correct distance to j.

So, what we have seen is that in negative edge weight case, what if we uses the strategy of Dijsktra's algorithm this may not be it is that we will burn a vertices just because it is shortest expected distance among the unburnt vertices does not guarantee that this is correct distances we might find later on a smaller distance. But, in spite of this, this update operation has some useful properties which we can expert.

(Refer Slide Time: 04:53)



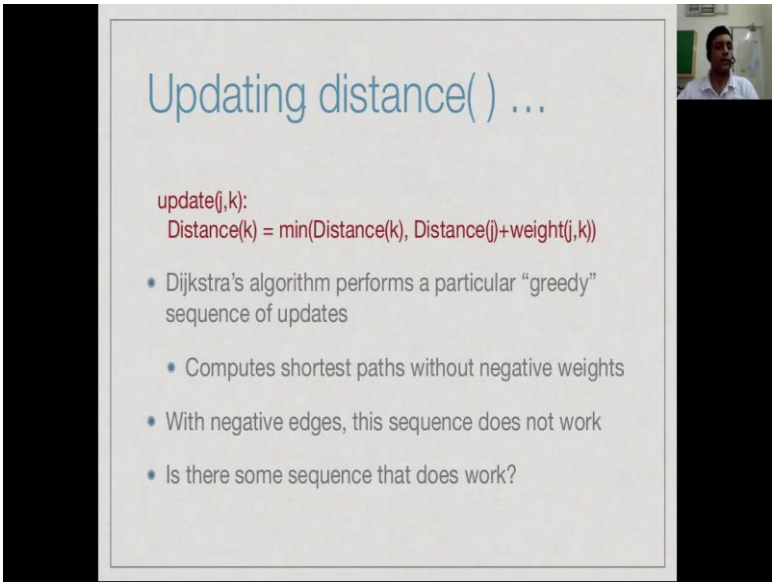
So, the crucial thing is that we update gives us some upper bound that the distance to k, we already have... So, at any point the invariant that we have is that the value that we have distances to k is greater than or equal to the actual shortest distance from the source to k, initially we may assume it is infinity. So, that is surely greater than the actual shorter distances. At any point when we reduce it, we reduce it because we have found a concrete path which gives us a shorter distance.

So, when we do this after this update we know that the distance to k is no more than the distance we have just discovered through j. Because, if it was already less and we keep it that way, because we found it through previous j prime or we have found it now in which case we have updated it on this update operation. This also means because of our previous observation about shortest paths that if actually on Dijsktra's algorithm also, that if distance j is actually correct and the shortest path now consists of just adding this edge. We are not claiming it with respect, if it is the shortest path, then the distance to k will be correctly computed.

So, in an actual this update operation is a safe operation, because it will never bring distance of k below the actual value. So, it will always be adopt the value that we want. So, we can keep doing Fourier updates, so unnecessary updates and it does not hurt us. So, redundant updates cannot accept this calculation, it just may not make progress, but may not it will not send us to a situation from which we cannot recover the minimum cost.

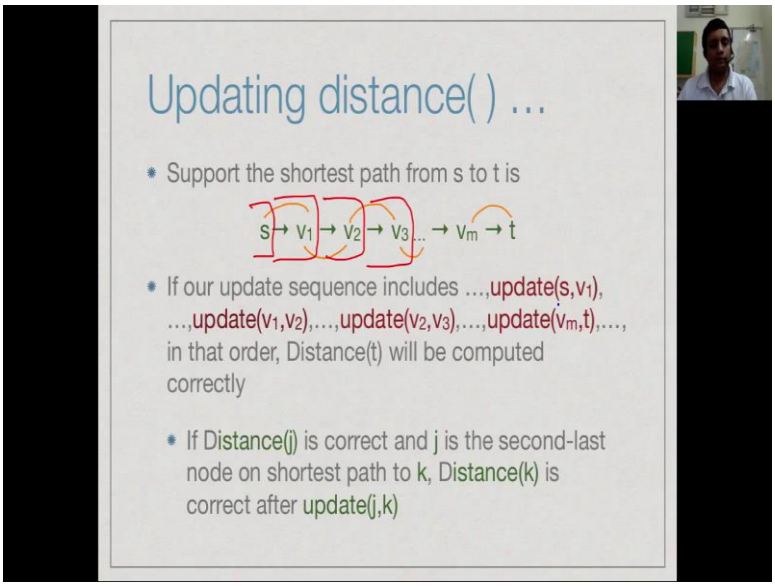
Because, we always be act or adopt the minimum cost, so whenever we do a min, we will always be coming down, but not processing below the actual value we want to find.

(Refer Slide Time: 06:29)



So, if you look at Dijsktra's algorithm then what it does is a particular sequence of greedy updates, it chooses the smallest distance vertex which is not burnt and un burns it and the proof breaks down, because this sequence does not match, necessarily give us a shortest path, if the ways can be negative. So, a natural question to ask is what sequence of updates should I do in order to actually get the shortest path? Is there a better way of computing the sequence of updates, rather than going via the 3D heuristics of Dijsktra's algorithm?

(Refer Slide Time: 07:10)

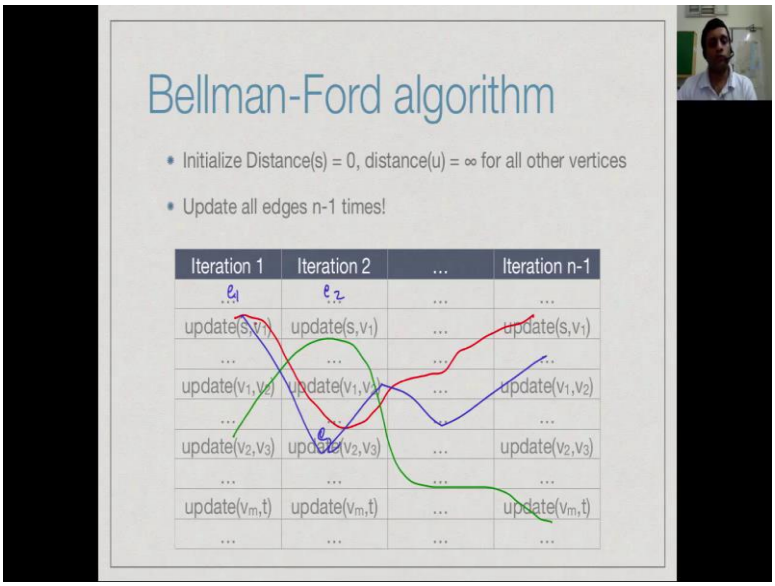


So, suppose we have two vertices s and t, we want to find the shortest path from s to t and suppose that is the shortest path which goes I have we want to v m we have seen before. Now, if we do the updates in this order that is we first compute the update from s to be 1 then we compute v 1 to v 2, then we compute v 2 to v 3 in between we can do a other things, it is only that these updates happen in this sequence then v 3 to v 4 and so on and finally, we do v m to t.

Then, what do you know where we knew earlier that if you have correctly computed the distance up to s, then this update will correctly compute the distance of v 1, because this is the second last node on this. So, if distance of j is correct and j is a second last node if the shortest path to k, distance of k is correct after the update, so because distance of s was 0 under to is correct, when we do update s v 1 this distance becomes correct.

Now, we might to also correct whether updates will do not bother us, but now when we come to v 2 we will be updating v 2 from v 1 and v 1 is a second last path on this shortest path. Therefore, this also becomes a correct value, then this also become the correct value. So, in the middle of all these updates if we identify this particular sub sequence of updates s to v 1 then v 1 to v 2 then v 2 to v 3 then we are what the distance. So, now the question is how do we make sure that we have this sequence of updates for s and t.

(Refer Slide Time: 08:38)



So, Bellman-Ford algorithm basically says do not write to your find the particular sequence, just generally compute all possible sequence have updates. So, at a high level this the algorithm it is says initially assigned the distance of s to be 0 and the distance of u to be infinity for every other vertex. Now, we just blindly do every updates n minus 1 times, so what we do is initially you update for every edge given the fact that s as distances 0 and u is infinity otherwise, then we update everything.

So, all are these updates where s is the source vertex and v is the target vertex, these will give us now some finite values for these means, where as these will just remain infinity. So, this will become finite and these will become infinity, now you done one full sequence of updates in which other then s all the neighbors are wish now some finite values, now you do this whole thing again.

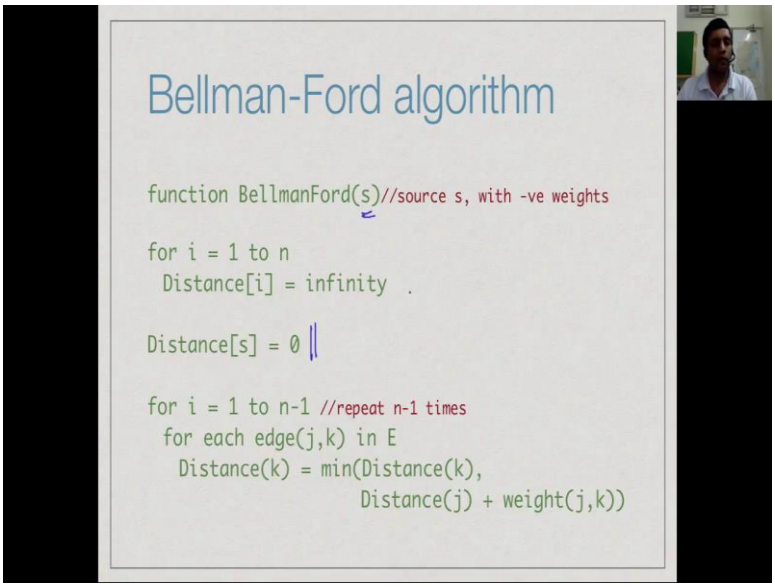
So, you update every a j k and then you update every a j k, now we know that a shortest path between any two vertices as at most n minus 1 edges. Therefore, if we do this thing n minus 1 time, the claim is that if I want any sequence of updates I might find the sequence of updates should was like this or a might find that any sequence of n minus 1 updates which is a legal path will be represented in this case.

So, this is a very clever matrix which has all the updates after one iterations, all the updates after two iterations or for want a particular sequence toward update e 1, then update e 2, then update then I will find it e 1 here, then I will look for e 2 here, then I will find e 3 here and then so on. So, every possible path is represented in particular given

average example here by s goes from the path goes from s to v 1, v 1 to v 2 and so on.

Then, we find that the first updates comes in iteration 1, then after have been updated v 1 the next iteration v 1, v 2 is was a correct distance to v 2, iteration 3 will give as v 2 to v 3 and so on. And finally, iteration n minus 1 actual it may not be n minus 1 is depends on the length of this path. So, when if this path actually as n minus 1 steps in the last one will give us the update from v m to t and the correct distance about t, but this could actually will be even less then n minus 1 it will path is of less lesser length.

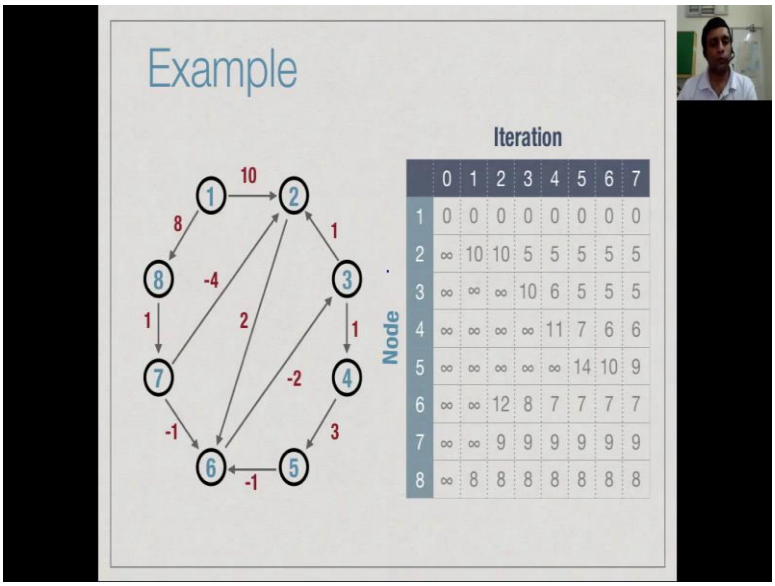
(Refer Slide Time: 09:55)



So, the algorithm is actually remarkably simple, do you start from a source vertex s. So, initially you assign the distance to be infinity for every vertex and you initialize the distance of s to be 0. Now, n minus 1 times you blindly repeat the following operation for every edge in your graph apply the distance updates. So, you take the distance of the target of that edge to be the minimum of the current distance of that target or the distance to the source of the edge plus the weight of the edge.

So, it just blindly to this n minus 1 times and it because of this property that you will find for every valid shortest path, you will find the sequence of updates which matches that path in this sequence that you are to computing here, you will always get the shortest path to every other vertex starting from s.

(Refer Slide Time: 11:44)



So, let us look at an example and see how the Bellman Ford algorithm actually works. So, here is a graph it has some cycles and it has an some negative weights, but there are no negative cycles, for instance we will see that here we have a cycles who weight is minus 2 plus 1 and in minus 1 plus 2 plus 1. Similarly, here we have a cycle whose weight is figure around from 6 to 3 to 4 to 5 we have minus 2 plus 1 is minus 1 plus 3 is plus 2 minus 1 is plus 1, so this is again cycle or plus 1.

So, there are negative weights, but no negative cycles and now in this we want to compute shortest paths from 1 to every other vertices. So, one is that source vertices, so we set up this iterations, so in the initial step we said the distance of vertix 1 to 0 and everything else as a infinity. Now, we will in the first step try to update all neighbors of things that we hear, now all values which are infinity do not update the neighbors, but the value 1 will it has 2 neighbors 2 only.

So, needs to as things now become finite values then an expected, now we have a these three vertices which are both finite values. So, you would expect that the neighbors namely 7 and 2 has a neighbors 6. So, the only out going is from 2 is 6 the only out going edge from 8 to 7, so you would expect 7 and 6 to get updated and indeed the next step you find the 6 and 7 get values which are updated from those values. So, 8 plus 1 9 10 plus 2 is 12.

Now, having got this you will find that we have now updated 1, 2, 8, 7 and 6, now notice that I now have another path not this path from 1 to 2 will have a path which goes this

way from 1 to 2 via 7 and this becomes a smaller path, because of the minus 4. So, in this iteration I will actually find the first non-trivial update which could not, which you bi late the Dijsktra's assumptions that all burnt vertices or invariant.

So, the distances two which was already computed 10 will actually shift, in addition to that of course, because we have new neighbors of 6 was 6 will now give us a new value for 3 and so on. So, in this iteration I find the reset, but the value of 2 reduces the value of three becomes something finite. Notice the value of 6 itself changes, why because earlier we will looking at a path like this and now we have discovered that this is actually a better path like this, which has also a negative edge to add to the benefit.

So, we come down from 12 to 8 plus 1 minus is 8, so we continual like this at the next step that path to 3 shrink further and this because now having get a better path for 6. So, in the previous path we said over the path for 6 plus 12 and 12 minus 2 was 10, now we said this go it is not 12 it is 8. So, 8 minus 2 is actually 6. So, you got a better path to 3 we are discovered in new path, because we had a path to 3 from 10 plus 1 is a 11 and so on.

So, we continue one more step and then because now we know that 4 was reachable and 11 steps 4 plus 3 is 14. So, 5 is reachable, but 4 itself now because the value of 3 got updated. So, in this iteration the value of 3 gets updated from 11 to 7, this will get propagate that after one more iteration. So, now this value comes down from 14 to 7 plus 3, but in this process what is happened is in the value of 3 as itself bound on one more time. So, int's value again reduces the value of 4 and then finally, after the n minus 1 iteration we get a stables set of values. So, this now turnout to be the shortest paths to all the vertices from this start vertex 1.