

IIT Madras
ONLINE DEGREE

Mathematics for Data Sciences 1
Professor. Madhavan Mukund
Chennai Mathematical Institute
Lecture No. 12.3

Single Source Shortest Paths with Negative weights

So, we are studying weighted graphs and we started looking at the shortest path problem, in particular, the single source shortest path problem and we saw Dijkstra's algorithm, and we said that it will work if we do not have negative weights. So, let us see what happens when we have negative weights.

(Refer Slide Time: 00:28)

Dijkstra's algorithm

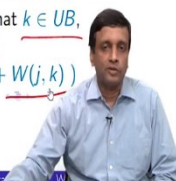
- Recall the burning pipeline analogy
- We keep track of the following
 - The vertices that have been burnt
 - The expected burn time of vertices
- Initially
 - No vertex is burnt
 - Expected burn time of source vertex is 0
 - Expected burn time of rest is ∞
- While there are vertices yet to burn
 - Pick unburnt vertex with minimum expected burn time, mark it as burnt
 - Update the expected burn time of its neighbours

Initialization (assume source vertex 0)

- $B(i) = \text{False}$, for $0 \leq i < n$
- $UB = \{k \mid B(k) = \text{False}\}$
- $EBT(i) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{otherwise} \end{cases}$

Update, if $UB \neq \emptyset$

- Let $j \in UB$ such that $EBT(j) \leq EBT(k)$ for all $k \in UB$
- Update $B(j) = \text{True}$, $UB = UB \setminus \{j\}$
- For each $(j, k) \in E$ such that $k \in UB$,
 $EBT(k) = \min(EBT(k), EBT(j) + W(j, k))$



So, first, let us recall Dijkstra's algorithm and look at it a little more formally than we did last time. So, remember that we thought of Dijkstra's as I build them operating in this burning pipeline story. So, we had these vertices as oil depots, and we had the edges as pipelines. And if we set fire to a source vertex, then the fire spreads at a uniform pace along all the pipelines, and then we try to calculate the order in which each of the vertices will catch fire and propagate the fire further along new edges.

So, in the process of doing this as an algorithm, what we do is we keep track of which vertices have already burned, that is the vertices for which we have already computed the shortest distance. And we have an estimate about how long it is going to take to reach the others so that we have an, what we call the expected burn time, which we keep updating as we go along.

So formally, we keep track of these two things, the burn status and the expected burn time. So initially, we set the burn status to be false. So, let us call it B of j . So, for every B of i for every vertex i , B of i is initially false. And just to keep track of an auxiliary quantity, we will let UB .

So, UB is just a set of unburned vertices here. So, UB is just stands for unburned. So UB is those k for which B_k is false.

So initially, UB is a set of all vertices. And the way we start this algorithm is we set the burning time of the source vertex which we are assuming to be 0 in this case. We assume that the burning time with the source vertex is 0 and everything else has vertex burning time infinity because we have no information. And then what we do is as long as there are unburned vertices, in this case, everything is unburned. So of course, there are unburned vertices, you pick up one of the unburned vertices, which has a minimum expected burn time.

So initially, there is only one that is the source. So, you pick up the source vertex, and you update its status, you update the status of the vertex, you pick up saying it is now burned. So, therefore, in some sense, its expected burn time is frozen as whatever it was now, and correspondingly, the unburned vertices will reduce because this particular vertex that you have chosen to burn now is no longer unburned.

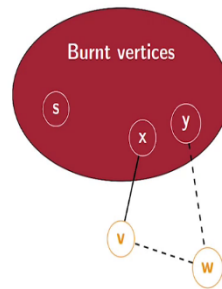
Now, more importantly, what you do is you look at every outgoing edge from this new vertex, so you just burn j . So, you look at every $j k$ edge and check if k is unburned, that is, we still have not fixed its distance, then you update its distance to be the minimum the distance you already know. So, this is what you already know about k plus the new information that you get, if you process this edge $j k$, which may or may not have been taken into account before obviously, because you have not reached this day before this. So, you look at what is the time that you burnt j and how much time it will take from j to k and this might well turn out to be smaller, so this is Dijkstra's algorithm.

(Refer Slide Time: 03:14)

Correctness requires non-negative edge weights



- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is v , via x
- Cannot find a shorter path later from y to v via w
 - Burn time of $w \geq$ burn time of v
 - Edge from w to v has weight ≥ 0
- This argument breaks down if edge (w,v) can have negative weight



So now, we look at this correctness proof and we argued why we need this edge weights to be non-negative. So, we said that we are incrementally discovering shortest paths. So, every new shortest path extends an earlier shortest path. And inductively, the burnt vertices are those for which that distance has been computed and frozen, we are never going to update those things again. So, at last, at some point in the algorithm, we have a big set like this, so we have this big set of burnt vertices.

So, all the vertices in this set, so we have the starting vertex, which now I have called s just to denote it and then we have various vertices x , y , and all that which we have learned so far. And what Dijkstra's algorithm now says is among the remaining, look for the one with the smallest expected burn time, and it will turn out that that will be connected to some vertex in the set, so it will be connected by an edge.

So, it will be the minimum of something plus the edge from x to v . And the argument was that if we now choose to burn v and add it to the set and freeze it is time at the current time, we will not be making a mistake and that is because if we find a new path to be later, it will come through another w , but that w must also start from inside the burnt vertices. So, when I burnt v , w had a higher expected burn time. So, otherwise, I would have been w .

So, when I get here, if I look at the cost of going to w plus the cost of going from w to v it cannot be less than what I already have for v and this crucially depends on the fact that this edge from w to v is not negative, because if it was a large negative edge, then going by a w , I could suddenly save a lot of cost when coming to v , so that is the crucial thing. So, the argument does not work, so we cannot freeze the cost of vertex the first time we burn it, if we are allowed

to revisit it by a negative edge later on. So, that is why Dijkstra's algorithm requires non-negative edge weights.

(Refer Slide Time: 05:11)

Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles
- Going around a cycle can only add to the length
- Shortest route to every vertex is a path, no loops

The diagram illustrates a graph with vertices s, x, y, v, and w. Vertices s, x, and y are grouped within a red oval labeled 'Burnt vertices'. Vertices v and w are shown as yellow circles. A solid line connects s to x, and a dashed line connects x to y. A solid line connects v to w, and a dashed line connects w to v, forming a cycle. The edges (v, w) and (w, v) are labeled with negative weights, represented by minus signs.

Madhavan Mukund Single Source Shortest Paths with Negative Weights Mathematics for Data Science

So, the difficulty is precisely this, that we stop considering updates once we burn a vertex. So, what happens if we start allowing updates even after we have burnt a vertex? So, then the notion of burning does not really make sense. So, this analogy that we have does not really make sense, but it is a plausible strategy and why is it a plausible strategy? This plausible strategy because though we are allowing negative edge weights, we do not have any negative cycles.

So, this means that if I am going from say, the starting vertex to some vertex x, it cannot help me to go through a loop because every loop is guaranteed to have a non-negative weight. Because there are no negative cycles. So, if I want the shortest path from any starting vertex, from the starting vertex to any other vertex x, I may as well assume that that path is really a path in the sense of, we have defined that is it has no loops, it has no repeated vertices.

(Refer Slide Time: 06:15)

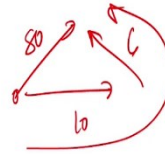
Extending to negative edge weights



- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{l-1}} j_{l-1} \xrightarrow{w_l} k$$

- Need not be minimum in terms of number of edges




So, if it has no repeated vertices, then what we can consider is in terms of the length of the path, not in terms of the weights of the path alone, but we have the shortest path, which takes us from say 0 (())(0:27) source vertex to a vertex k . So, not only is the weight minimum, but the reason that this is the path I chose is because there are no shorter paths which have the same weight or less, so it need not be the shortest path overall, we saw examples where you could have one edge, which takes me with 80. But then if I take two edges, I might go with 10 plus 6 16.







So, the shortest path might well be a roundabout path. But what we are saying is that if there are, there is a path of length 2 which is shortest there is no path of length 1, that is what this means. So, this is the, so 1, if I have to do this, I have to take these 1 steps in order to get to k , and there is no better way of getting to k to achieve this cost. So, this is not the minimum number of edges going from 0 to k without considering weights, but it is the minimum number of edges and the shortest weight if you consider weights.

(Refer Slide Time: 07:14)

Extending to negative edge weights



- Suppose minimum weight path from 0 to k is
$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$
 - Need not be minimum in terms of number of edges
- Every prefix of this path must itself be a minimum weight path
 - $0 \xrightarrow{w_1} j_1$
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
 - ...
 - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{\ell-1}} j_{\ell-1}$
- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k
- Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
 - Update cannot push this distance below actual shortest distance
- After ℓ updates, all shortest paths using $\leq \ell$ edges have stabilized
 - Minimum weight path to any node has at most $n-1$ edges
 - After $n-1$ updates, all shortest paths have stabilized



Madhavan Mukund

Single Source Shortest Paths with Negative Weights

Mathematics for Data Science

14

Now here, we again go back to our old argument, which says that, okay if this is my path, then what happens when I come to j_1 , could I have come to j_1 any better than using the weight w_1 ? Well, if I could have come by a different route and come less than w_1 , then I could come to j_1 and then continue with the same path to k , so if I could come to j_1 earlier than I have now, then I could use that path plus the path from j_1 to k and get a shorter path to k .

So, if this is the shortest path to k , then this must also be the shortest path to j_1 , the shortest path to j_2 , the shortest path to j_3 , and all that. And so, every prefix of this path has to itself be a shortest path. So, this gives us a starting point to think of an algorithm to deal with negative edge weights. So, in some sense, once we have updated this one, once we have found the shortest path to j_1 minus 1 by some algorithm, then we know that the update that we get for k is going to be a final update, there is not going to be a better one than that, because, if I keep decreasing, I can only decrease up to the shortest path, I cannot go below the shortest path.

And when will I hit the shortest path when my nearest neighbour which feeds that shortest path is also frozen. So, in some sense, if my neighbour's shortest path is known, then my shortest path will be known in the next step. So, therefore, for that neighbour, that neighbour's shortest path should have been known in the previous step. So, we can try to see if we can fix these shortest paths one at a time, if I can fix j_1 then I can fix j_2 , we can fix j_2 , then I can fix j_3 , and so on.

And once I fix j_1 minus 1, then I can fix k . So, what we want really is an algorithm, which tells us that after we have done l updates, so an update for us is what we did when we burnt a vertex, when we burnt a vertex we reset the burning time to be the minimum of what we already had

plus the new burning time we discovered through the recently discovered vertex. So, if we can guarantee that after we have done this 1 times, we have made sure that there are all paths of length 1 are at a minimum, there are no shorter weight paths of length 1, then since there are at most n minus 1 edges, then if we do this n minus 1 times there are no paths without repeating vertices and no strong notion of a path there are no paths which have more than n minus 1 edges in them because once you take the n th step, you have to repeat a vertex.

So, if we have this property that we can update and make sure that after l updates, all the paths of length l in terms of number of edges have achieved their minimum that is there are no shorter ways to go l edges or less, you have to take maybe one more edge and take a negative edge that is different, but you cannot get that l edges. This is then we can guarantee that we can see sort of first find all shortest paths.

So, this is like a combination of breadth-first search which finds shortest paths by length of path, and our weighted thing which finds it by length of weight. So, what we are saying is that up to this path length in terms of edges, there are no shorter paths in terms of weight. And this is the kind of property that we want.

(Refer Slide Time: 10:14)

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
 $D(k) = \min(D(k), D(j) + W(j, k))$

Handwritten diagram: A vertex j has an arrow pointing to vertex k . Above j is $D(j)$ and above k is $D(k)$. A red circle around the update formula $D(j) + W(j, k)$ has an arrow pointing to $D(k)$.

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science | IIT Madras

So, this algorithm is called the Bellman Ford Algorithm. So, it is a much simpler algorithm, in some sense, than Dijkstra's to think of, although it requires a little bit of understanding like we just did to see why it works. So, all we do is we like Dijkstra, this expected burn time, so we keep track of the distance to every vertex as far as we know so far. So initially, the distance to the source vertex, which again, we assume is 0, the source vertex is 0, and its distance is 0, and everybody else distance is assumed to be infinity.

And now comes the update. What Bellman Ford does is just n minus 1 times it just blindly updates everything. So, it takes every edge and it looks at the starting point at the edge and the ending point of the edge. So, I have an edge, which goes from say, some j to some k . So, there is currently at this iteration, there is some distance that I have associated with j and then I have associated some distance with k .

And I have what I would get if I take this edge and append it to j . So, this is a candidate to replace D of k . So that is what we do, we just check for every k , whether the weight that the distance that we are currently assumed for k , is it smaller than the distance that I would get if I take one of my neighbours and add that edge weight from that neighbour to that distance. So, I just blindly do this n minus 1 times and the claim is that this will give you the shortest path.

(Refer Slide Time: 11:48)

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \dots, n-1\}$,
for each edge $(j, k) \in E$,
 $D(k) = \min(D(k), D(j) + W(j, k))$

Works for directed and undirected graphs

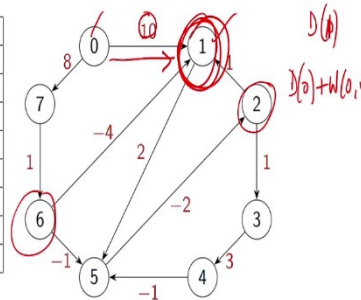
The slide also features a graph with 8 vertices (0-7) and weighted edges: (0,1): 10, (0,7): 8, (1,2): 1, (2,3): 1, (3,4): 3, (4,5): -1, (5,6): -1, (6,7): 1, (7,0): -4, (1,5): 2, (5,1): -2. A video inset shows a lecturer, Madhavan Mukund, speaking.

So, this works for both directed and undirected graphs. So, the example we did for Dijkstra's algorithm was for undirected graphs, but you could as well do it for directed graphs. Because anyway, we are following edges in one direction only when we compute the shortest thing. So, let us look at this example, so this is a directed graph, it has some negative edge weights, like minus 4, minus 1, and there are arrows in this, so there are some so you can go for example, from 0 to 1, but you cannot come back from 1 to 0, and so on. So, let us see how this Bellman Ford algorithm would work on this.

(Refer Slide Time: 12:18)

v	$D(v)$						
0	0						
1	∞						
2	∞						
3	∞						
4	∞						
5	∞						
6	∞						
7	∞						

■ Initialize $D(0) = 0$



So, what we do is we keep recomputing this distance D of v . So, remember that D of v is the best distance I know of right now for vertex v . So, in this case, there are 8 vertices from 0 to 7. And we are going to iterate this thing after initialization, n minus 1 times. So, there are 8 vertices, so n is 8. So, we are going to run this algorithm, this iteration 7 times, and then 7 times, it should hopefully give us the shortest path from 0 to every other vertex.

So, we initialize it by setting the distance of 0 to the vertex 0 to 0, and everything else to infinity and this is our initialization for Bellman Ford. So initially, we know nothing about how to get to any other vertex. And now we do this update. So, now we look at every vertex or we look at every edge is how the Bellman Ford algorithm says, we look at this edge and we say, what do I know about the starting point plus this weight versus the ending point?

So, the update is compare D of 0 to D of 0, D of 1 to D of 0 plus the weight of 0,1. So, what should I put here, is a question. So, should I leave it as what it is, or should I update it by some new information that I have got about the edge coming into it. Of course, I could do it for D of 2 also, but then I know that for D of 2 and D of 6, nothing has happened because everything is infinity, so is really D of 0, which carries some importance at this stage.

(Refer Slide Time: 13:42)

Bellman-Ford Algorithm

v	D(v)			
0	0	0		
1	∞	10	✓	
2	∞	∞		
3	∞	∞		
4	∞	∞		
5	∞	∞		
6	∞	∞		
7	∞	8	✓	

- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(D(k), D(j) + W(j, k))$$

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science

So, if we do this, we find that from 0 I can get to 1, and from 0 I can get to 7. And therefore, the entries for 1 and 7, get updated from infinity, which is what I knew before to 0 plus 10, in the case of 1, and 0 plus 8 in the case of 7. So, I updated to the time to reach 0 plus the weight of the edge from 0 to that vertex and everything else is infinity because I cannot reach it from 0 at this point. But now, I have some information about 0, 1, and 7 so, in the next step, I can look for any vertex, which is either connected to 7 or connected to 1. So, not that one, but say this one. So, 1 is connected to 5.

(Refer Slide Time: 14:25)

Bellman-Ford Algorithm

v	D(v)			
0	0	0	0	
1	∞	10	10	
2	∞	∞	∞	
3	∞	∞	∞	
4	∞	∞	∞	
5	∞	∞	12	
6	∞	∞	9	✓
7	∞	8	8	

- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

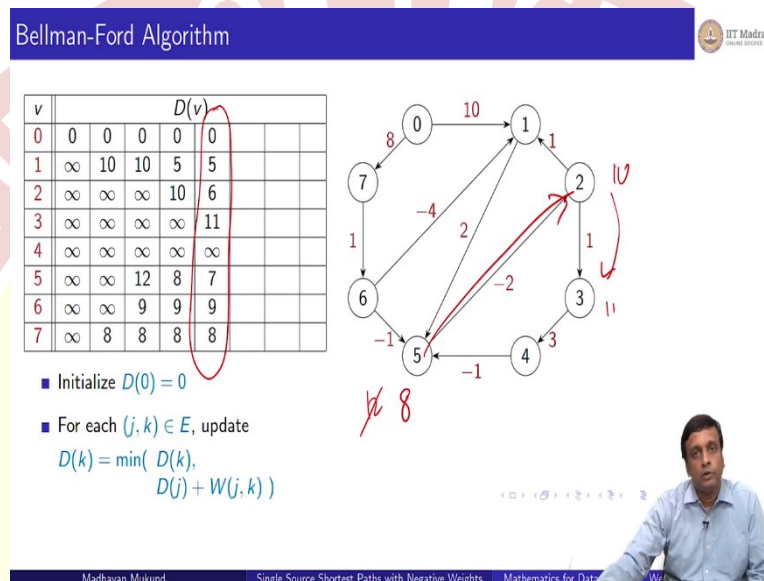
$$D(k) = \min(D(k), D(j) + W(j, k))$$

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science

So, in the next step, what happens is that because 1 is connected to 5 and I know that I can reach 1 at time 10, then in 10 plus 2 I can reach 5 in time 12 earlier I believe it was infinity.

So, I can replace it by 12. In the same way, if I look at 6 for instance, earlier I thought it was infinity but now I know that I can reach 7 and time 8, and 7 plus 1 is 9, 8 plus 1 is 9. So, therefore, I can reach vertex 6 in time 9. So, these two things which are connected to the vertices are recently burned, get updated. So, we keep doing this. So, now we have burned, 6, and 5, not burned but we have updated 6 and 5 in addition to 0, 1, and 7. So now, we will find new paths because 6 also has outgoing edges.

(Refer Slide Time: 15:13)



So, in particular now, what we will find is that there is a strange phenomenon that we have discovered, which is that if I come from 6, so remember that 6 was, had been assigned 9 before, so, right now these are the numbers that we have everything else is infinity. So, now, if I am at 9, and if I take this negative edge, then I can come from 0 to 1 at cost 5. So, instead of going directly in cost 10, which is what I had earlier assumed, I could take this roundabout route, and I could do 8 plus 1 minus 4, and come there and 5.

So, this is the kind of update that Dijkstra's algorithm would not have discovered it because it would have really frozen that thing at some point, saying that it is already given, and therefore, I will not update. So, this 10 becomes 5, what about 2? Well, now 2, I can reach from 5. So, it is 12 minus 2, so this becomes 10. And 5 itself, now is interesting, because earlier, I had to come this way. And that was costing me 12. But now because I can come from 6 directly, I can come to 6 and 9, and then 6 to 5 will give me so this has become 8.

So, in this way, we keep updating, so now after this, now that I know 2, I can even update 3 because 3 is reachable from 2. So, if I can reach 2 in time 10, I can reach 3 in time 11. But of course, 2 itself has now got a better route. Because having come to 5 in time, 8 now.

Remember, it was 12 and then it became 8. Now I can go from 5 to 2 in times 6. So, each time I am looking at the previous row, so the fact that 2 gets updated from 10 to 6 now does not yet reflect in the fact that 3 should be updated, so 3 is updated from infinity to 11 because I knew that 2 was 10 before I did.

So, all these updates are happening at this time based on the previous times information. So, I am not doing it in sequence in that sense, so though I calculate that the distance to 2 is 6, I do not use it to calculate the distance to 3 is 7 yet, I will do it in the next round.

(Refer Slide Time: 17:28)

Bellman-Ford Algorithm

v	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(D(k), D(j) + W(j, k))$$

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science

Bellman-Ford Algorithm

v	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(D(k), D(j) + W(j, k))$$

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science

So, in the next round, I discovered that 2 was frozen at 6, and therefore now D has become 7. And finally, I have also found something that reaches vertex 4 because now I have got this path which goes this way or there are other paths also. So, we have this path also, which goes this

way, and so on, so we have many paths which come to 4, but I only now reach that and I calculate 14. So, I keep iterating this, and I get slightly better paths everywhere.

And finally, after I have done this 7 times, I have discovered a stable thing and you can calculate that if you do this one more time you should get no updates. There should be no shorter paths if there are no negative cycles. So, this is how the Bellman Ford algorithm works. It just keeps updating every vertex every time and you do it a fixed number of times, which is the number of vertices in your graph, minus 1. And once you have done that, you are guaranteed that all the paths have stabilized.

(Refer Slide Time: 18:22)

Bellman-Ford Algorithm

v	D(v)							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

- What if there was a negative cycle?
- Distance would continue to decrease
- Check if update n reduces any $D(v)$

Madhavan Mukund | Single Source Shortest Paths with Negative Weights | Mathematics for Data Science | IIT Madras

So, what would happen if there was a negative cycle? Well, the path would not stabilize, there would be a way to take a path longer than n , n minus 1 edges go beyond that and go around the cycle and get still shorter. So, if I iterate this one more time, and the distances decrease, then I know that there is something wrong. So, this is one way. So, either you can assume there are negative cycles and keep running it or you can run it.

And then when you come to n th iteration, which you normally should not need, you need to stop with n minus 1, but you can run it one extra iteration. And see if you get a decrease in the distance. And if you get a decrease in the distance, that means there was a negative cycle. So, check that the n th update should not reduce any D of v . If it does not reduce once, then after that, once the D of v is stabilized, there is going to be no update because the previous update is just going to propagate. So, once I get the column repeating, there is going to be no change further on. So, if the column changes at the n th step, then you know that you had a negative cycle.

(Refer Slide Time: 19:19)

Summary



- Dijkstra's algorithm assumes non-negative edge weights
 - Final distance is frozen each time a vertex "burns"
 - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length $1, 2, \dots, n-1$
- Update distance to each vertex with every iteration — Bellman-Ford algorithm
- If Bellman-Ford algorithm does not converge after $n-1$ iterations, there is a negative cycle

Navigation icons



Madhavan Mukund

Single Source Shortest Paths with Negative Weights

Mathematics for Data Science

So, we saw Dijkstra's algorithm and Dijkstra's algorithm assumed non-negative weights and the reason that we needed that property was because of this strategy we used to freeze the distance to burn vertices once they were burned. So, we never looked for updates to that. So, we should not have found any new updates through negative edges. Otherwise, that strategy is not correct.

But assuming that we do not have negative cycles, remember if we have negative cycles, the notion of a shortest path is not defined, because you can go round and round the cycle and you can make a shortest path as short as you want. So, the shortest path is defined only when there are no negative cycles, even if there are negative edge weights. And in such a case, what we just said is that the shortest path is a path, it cannot involve a loop and every prefix of that shortest path is also a shortest path.

And you can use this in this Bellman Ford algorithm to iteratively find the longest, the shortest paths of length 1, of length 2, length 3, and so on. And after n minus 1 iteration, you have automatically found shortest paths to everything. So, in a way, Bellman Ford is a much simpler algorithm to think of, it is just a blind iteration, which is n minus 1 times you keep updating it, you do not have to keep track of anything that was burned and you do not have to keep track of expected burn time separately and all these things. You just keep updating every time you make an update, you look at all the neighbours and update them again.

And the property of this, this fact that the shortest paths are monotonic in the sense that every path is an extension of a shortest path guarantees that after n minus 1 steps all these updates will converge unless you have a negative cycle. So, you can also use this algorithm to find

negative cycles in that sense, if you go through this whole process and you do it one more time and you find a decrease, then you have a negative cycle.

