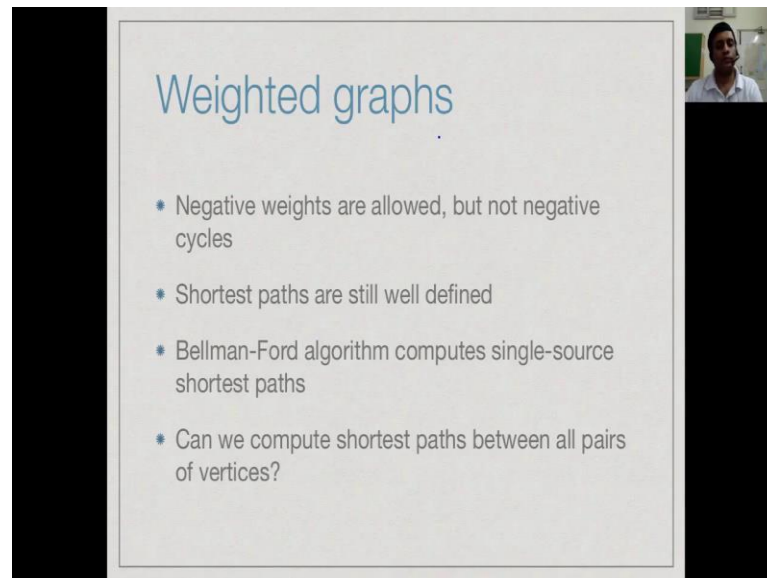


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 03
Lecture - 28
All-pairs Shortest Paths

Let us now turn our attention to the All-pairs Shortest Paths problems, where we try to find the paths, shortest paths between every pair of vertices in a graph.

(Refer Slide Time: 00:10)



Weighted graphs

- Negative weights are allowed, but not negative cycles
- Shortest paths are still well defined
- Bellman-Ford algorithm computes single-source shortest paths
- Can we compute shortest paths between all pairs of vertices?

So, recall that we are working with weighted graphs, we allow negative edge weights, but not negative cycles, because with negative cycles our shortest path is not well defined, with negative weights, it is well defined. We saw that the Bellman-Ford algorithm allows us to generalize Dijkstra's algorithm and compute single source shortest paths in weighted graphs with negative weight, but without negative cycles.

So, now, we want to further generalize this and compute not just the shortest paths from a single source, but the shortest path between every pair of vertices. So, as we explained that the beginning an example would be, if you trying to run a travel website or an airline website and somebody wants to find the minimum cost or the minimum time flight or train between any pair of sides.

(Refer Slide Time: 00:54)

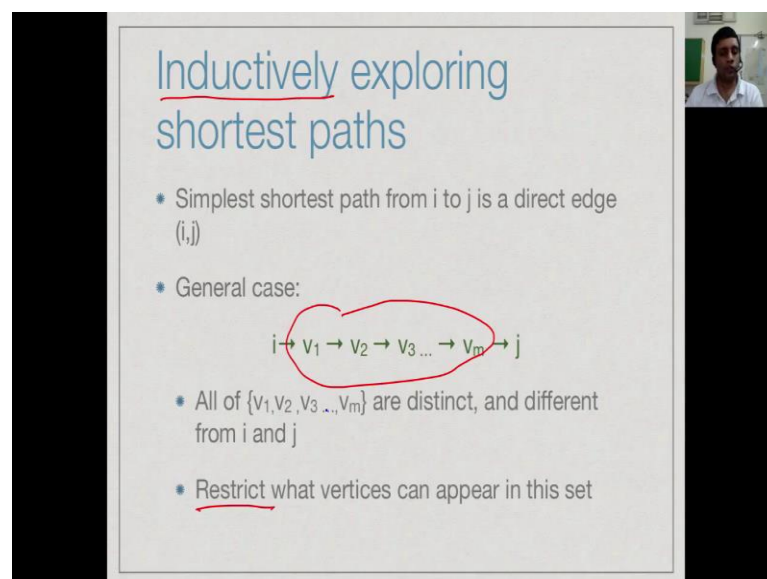


About shortest paths

- Shortest paths will never loop
- Never visit the same vertex twice
- At most length $n-1$
- Use this to inductively explore all possible shortest paths efficiently

So, you made the following observation of shortest paths that the shortest path even in the presence of negative weights will never loop, because we can always remove the loop without increasing the length of the path. So, therefore a shortest path never visits the same vertex twice and has length at most n minus 1. So, we exploited this algorithm in one way for Bellman-Ford and we will find that we can use it now for an inductive algorithm for all pairs shortest path. So, we will come up with an inductive solution of how to build up the shortest paths.

(Refer Slide Time: 01:30)



Inductively exploring shortest paths

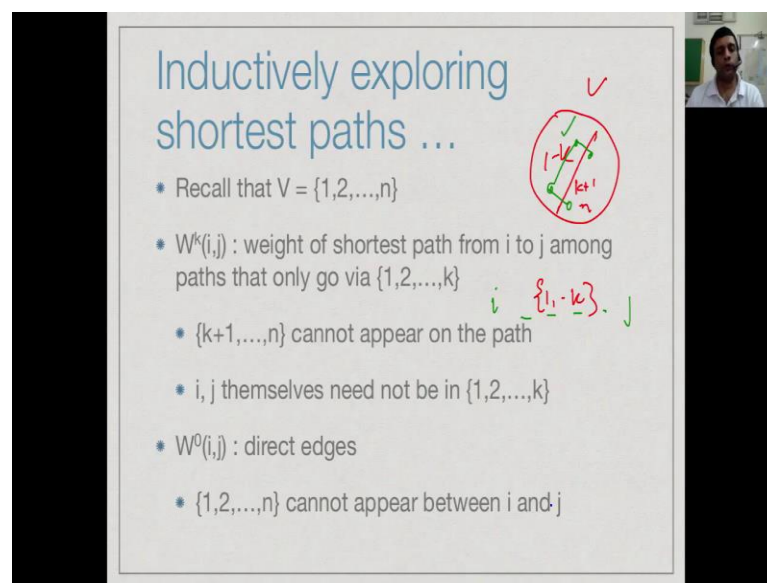
- Simplest shortest path from i to j is a direct edge (i, j)
- General case:
$$i \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \dots \rightarrow v_m \rightarrow j$$
- All of $\{v_1, v_2, v_3, \dots, v_m\}$ are distinct, and different from i and j
- Restrict what vertices can appear in this set

So, we are basically going to build up shortest path in terms of vaguely by length, but most specifically what vertices we allowed in between. So, the simplest shortest path you

can imagine in a pair of vertices it is just a single edge. So, we have single edge and this happens to be the shortest path, then you are in good shape. But, in general this may not be the shortest path, because even if there is such an edge, there may be a way, because of negative, because of edge way, even because of negative edge ways are reaching from i to j by our longer path of edges, but to the shorter overall cost.

But, what we do know, because of the characterization of shortest path is that this path from i to j goes through some intermediate vertices all of which are different from each other, no vertex is visited twice. And secondly, none of these vertices either i or j , there is no point in coming back to i and then going to j , so there is no route anywhere in this path. So, what we will do for this inductive thing is to restrict, what can happen in between i and j , what are the vertices that are allowed and we will gradually increase the set. If we allow any vertices, then of course, we will get arbitrary shortest paths.

(Refer Slide Time: 02:40)



Inductively exploring shortest paths ...

- Recall that $V = \{1, 2, \dots, n\}$
- $W^k(i, j)$: weight of shortest path from i to j among paths that only go via $\{1, 2, \dots, k\}$
- $\{k+1, \dots, n\}$ cannot appear on the path
- i, j themselves need not be in $\{1, 2, \dots, k\}$
- $W^0(i, j)$: direct edges
- $\{1, 2, \dots, n\}$ cannot appear between i and j

So, recall that vertices are numbered 1 to n , so we will compute a quantity W^k of i to j to be the weight of the shortest path from i to j , where we restrict the vertices that can be used go from i to j to be between 1 and k . In other words, we have this set of vertices V , so we cut it off saying we have 1 to k and we have $k+1$ to n , we say that only these vertices can be used in the path.

Now, the end points themselves may not be, need not be, but at the point is that in the end point is outside, then it can still, so I could have a 1 vertex here and then it could have a path which go like this and then come back. So, only says is that the intermediate

vertices, so when I start with i and go to j , what happens in between lies in this set 1 to k . So, in particular if k is 0, because our numbering is 1 to n , it says that the vertices that can appear cannot include 1.

So, in other words, if I have W^0 , then it says all of the vertices 1 to n cannot appear on the paths, so the only way that we can find such a shortest path is to the direct edge. So, W^0 , the base case of this inductive definition says that the shortest paths between i and j which exclude all vertices from 1 to n are of the form edges between i and j .

(Refer Slide Time: 04:07)

Inductively exploring shortest paths ...

- From $W^{k-1}(i,j)$ to $W^k(i,j)$
 - Case 1: Shortest path via $\{1,2,\dots,k\}$ does not use vertex k
 - $W^k(i,j) = W^{k-1}(i,j)$
 - Case 2: Shortest path via $\{1,2,\dots,k\}$ does go via k
 - k can appear only once along this path
 - Break up as paths i to k and k to j , each via $\{1,2,\dots,k-1\}$
 - $W^k(i,j) = W^{k-1}(i,k) + W^{k-1}(k,j)$
- Conclusion: $W^k(i,j) = \min(W^{k-1}(i,j), W^{k-1}(i,k) + W^{k-1}(k,j))$

So, now since this is an inductive definition, what we have to say is, supposing we know the shortest paths which use 1 to k minus 1, then how do I compute the shortest paths reduce 1 to k . So, we know among all the paths which use at most 1 to k minus 1, what is the shortest path from i to j , how do we now compute, if we allow vertex k also to be used, how do you compute, what do be the shortest path from i to j .

So, there are two cases, the first case is this is an extra vertex case not useful, the shortest path even if I include k does not use vertex k , it is enough to use 1 to k minus 1. In this case the shortest distance from W i to j using 1 to k is the same as the shortest distance from i to j using k minus 1. So, I can say W^k of i j is the same as W^{k-1} i j . On the other hand, it could be that using k does give us some non trivial improvement.

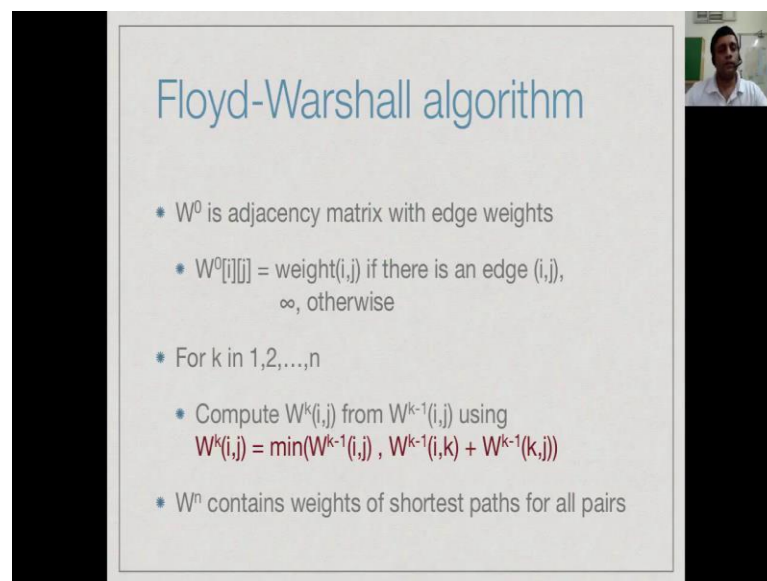
So, we have now some path which goes from i to j and on the way it visits k . So, if it visits k on the way, then we can break it up as a path from i to k and a path from k to j . But, notice that we have already said that this vertex k or any vertex for that matter

which appears in this path, if it is the shortest path appears only once. So, if cut k appears here, there is no k between i and k and there is no k between k and j . This means that I have a path from i to k already which goes through 1 to k minus 1 and I have a path from k to j already which goes 1 to k minus 1 and I am combining these two.

So, I can break up the path, it is the path from i to k and the path k to j , each of which uses only 1 to k minus 1 . And what is the cost of that path, well, we inductively know that we have the cost of the best path from i to k using only 1 to k minus 1 inductive k minus 1 , you also have the best path from k to j in our matrix W^{k-1} . So, if I add these two, this must be the path best way of going i or k .

So, combining the cases, we say that either we do not use k in which case, we take the value of the old matrix or you do use k and which case we combine two entries going i or k in the old matrix, we will take the smaller of these two. So, which one of these smaller will be the correct value of the shortest distance from i to j going through 1 to k .

(Refer Slide Time: 06:27)



Floyd-Warshall algorithm

- W^0 is adjacency matrix with edge weights
 - $W^0[i][j] = \text{weight}(i,j)$ if there is an edge (i,j) ,
 ∞ , otherwise
- For k in $1, 2, \dots, n$
 - Compute $W^k(i,j)$ from $W^{k-1}(i,j)$ using
 $W^k(i,j) = \min(W^{k-1}(i,j), W^{k-1}(i,k) + W^{k-1}(k,j))$
- W^n contains weights of shortest paths for all pairs

So, this gives us an immediate algorithm which is called the Floyd-Warshall algorithm. So, we start off with matrix representing the function W^0 . So, W^0 has entries which are exactly the edge weight, so there is an edge from i to j , the W^0 i to j says that direct path of that weight. Because, remember that W^0 cannot go through any intermediate paths and if there is no edge, since I cannot go through any intermediate vertex, W^0 i to j must be infinity.

Now, for k in 1 to n , I basically repeat this n times, I first allow 1 to be use. So, I

compute W_1 and W_0 . And how do I do that, I use the update to insert earlier that will take the minimum of what you already have plus the possibility going through the newly introduced vertex, then I will allow 1 and 2, then I will allow 1, 2 and 3. And obviously, if I allow 1, 2, 3 up to n I allow all the vertices to appear in between, W_n will now have the shortest paths with no constraints.

So, I need to do these update exactly n times, so that after n times I capture the shortest way paths which include any arbitrary combination of vertices on the path.

(Refer Slide Time: 07:37)

Floyd-Warshall algorithm

```

function FloydWarshall
  for i = 1 to n
    for j = 1 to n
       $W[i][j][0] = \text{infinity}$ 
       $W[i][j][0] = \text{weight}(i,j)$ 
  for each edge (i,j) in E
     $W[i][j][0] = \text{weight}(i,j)$ 
  for k = 1 to n
    for i = 1 to n
      for j = 1 to n
         $W[i][j][k] = \min(W[i][j][k-1], W[i][k][k-1] + W[k][j][k-1])$ 
  
```

Handwritten notes on the slide:

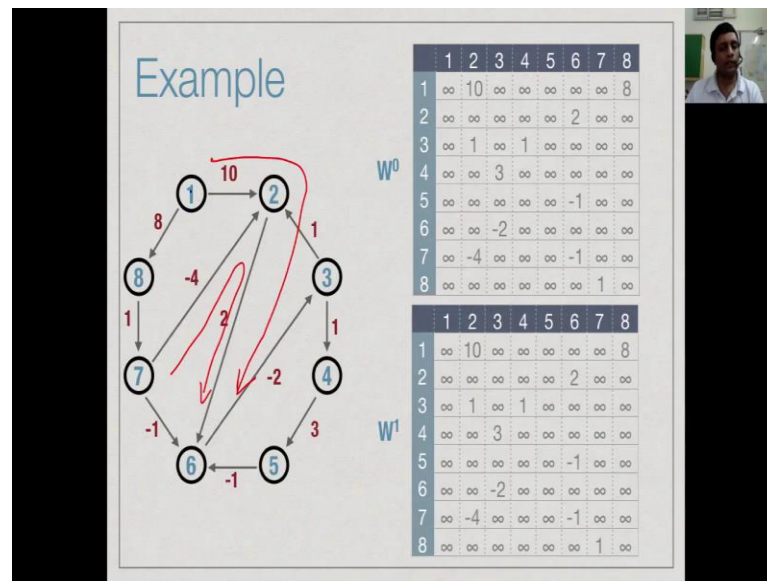
- $W^0(i,j) \leftarrow \text{weight}(i,j)$ (with ∞ below it)
- for $i = 1$ to n , for $j = 1$ to n

So, the actual code is again like Bellman-Ford extremely straight forward. You just have first initialization which sets every way to infinity and then updates the non trivial weights for those edges which had the graphs. So, we have keeping track of this function W_0 of i, j by a three dimensional matrix, so i and j represent the two vertices and the 0 represents the iteration number.

So, initially W_0 of i, j is either the weight of i, j , if there is an edge or it is infinity and that is what these first two steps handling. And now, we do this ends n times, we do this iteration, that is we update all the $W_{i,j}$'s at level k to be the minimum of the $W_{i,j}$'s at level k minus 1 or the sum of the root. So, implicitly here should have been for ((Refer Time: 08:33)). So, this is for i, n , for i equal to 1 to n , for j equal to 1 to n .

So, we have this update rule and we do these blindly n times and n we claim that the matrix W in the entry k has got the correct shortest paths for all pairs of vertices.

(Refer Slide Time: 08:55)

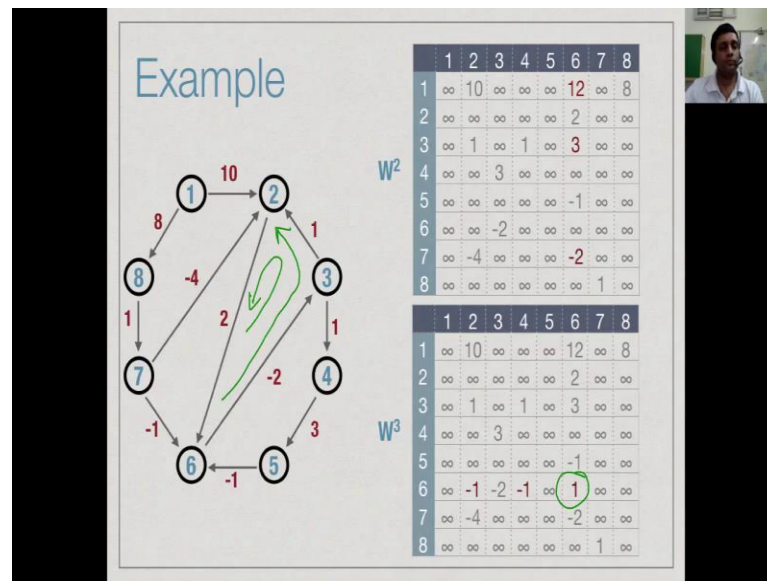


So, let us look at the same example that we say for Bellman-Ford. So, initially we assign to each, so W^0 has the edge weight. So, we have an adjacency matrix like representation in which for instance, we say 1 to 2 there is an edge of weight 10, 7 to 6, there is an edge of weight minus 1 and so on. So, we just duplicate all of these edges in the matrix and everything else is left at infinity.

Now, from W^0 , we go to W^1 by considering all new paths that we can find by going from 1, but now notice that this nothing that is coming into 1. So, this is signified to in fact, that the column 1 has infinity, no edge goes into 1. So, no other vertex can use the fact that 1 is connected to 2 and 8. So, if you update using update rule, you find that the W^1 is actually equal to W^0 nothing changes, because allowing 1 to be use in between two vertices does not help us.

It is not in between any two vertices, I cannot go from anywhere to 1 and then from 1 to that vertex. On the other hand, if I can now include 2, then I can do interesting things, I can go for example, from 1 to 2 to 6 and I can go from 7 to 1 to 6. So, align 2 and 1 as an intermediate vertex, so 1 does not help right now, but allowing 2 gives me something. So, if I now look at W^1 , I do not need W^0 anymore, whenever W^1 , then I will whether said be able to explore paths through 2. So, in particular 2 goes to 6, anything pointing into 2, so 7 to 2 to 6, 3 to 2 to 6 and 1 to 2 to 6, these three entries will get updated.

(Refer Slide Time: 10:13)



So, 1 to 2 to 6, so I get a new entry 12, 3 to 2 to 6, so I get new entry 3 and 7 to 2 to 6, I get minus 4 plus 2 minus 3. Now from W^2 , likewise I will compute W^3 , again compute W^3 will come back to this later, I do not need W^1 , I only need W^2 . So, I can throw away W^1 from now, I am just look at W^2 . So, now I am allowing myself to use 1, 2, 3, so I will look for things at go throw 3.

So, for instance now throw 3, I can go from 6 to 4, for example, so I will get entries of that form. So, I can go from 6 to 4 with new thing, I can also go from 6 to 2. So, now, I have a new way are going from 6 to 2, so that also get separated. And interestingly, I also discovered that, there is now our path from 6 to 6, because earlier I did not know that, but now that, I am allow to go throw both 2 and 3, I can go from 6 to 6, I have discovered this loop, which has a positive weight.

If did not have a positive weight viewer in trouble, because this is did not have a well defined solution and so on. So, you can just keep on doing this, we will not update up to W , if you now go all the way and do up to W^8 , then after you allow everything from 1 to 8 to be an intermediate thing, this matrix will actually compute all pair shortest path between any i, j .

(Refer Slide Time: 11:52)

Complexity

- Easy to see that the complexity is $O(n^3)$
- n iterations
- In each iteration, we update n^2 entries
- A word about space complexity
 - Naive implementation is $O(n^3) - W[i][j][k]$
 - Only need two "slices" at a time, $W[i][j][k-1]$ and $W[i][j][k]$
 - Space requirement reduces to $O(n^2)$

Handwritten notes: *Generalizes Bellman-Ford* (green), $O(n^3)$ (blue), $O(mn)$ (blue), $n.n.n$ (blue).

So, this algorithm it is very easy to see that the complexity is order n cube, because you have n iteration and an each iteration we are updating the entire matrix which has n square entries. This is not much you can do to improve this, because it is an adjacency matrix base algorithm, we cannot move to list, we do not have to compute any minimum maximum.

So, nothing much you can do it is, it is n cube algorithm, notice that, it is sort of sounds as it trivial case, the Bellman-Ford, because once you computed all pairs, so this generalizes as the solution Bellman-Ford. Because, in particular, if you now want all the shortest path from a given S , everything you just have to look up that particular row in the Floyd Warshall matrix.

You look at the row S and all the entries with say the shortest path from S , but remember that in that particular case, if I only wanted from S Bellman-Ford with clever adjacency list representation would take order $m n$, but it is here this will require order n cube. So, this is the generalize Bellman-Ford instance, you get same answer that you would have call from Bellman-Ford and more.

But, you are always spending n cube time, there is if you had very few edges in your graph which is typically the case Bellman-Ford in more efficient. So, if you are using, if you only want to do single source shortest path, you should not typically jump directly to Floyd Warshall, you should probably do Bellman-Ford instant. About space complexity, so we said that we are going to represent each $0 W 0, W 1$ etcetera is one coordinate.

So, we have basically have n times, n times n , because we have n times n is the actual matrix, we have n of these matrixes, because we have level 0, level 1 and level k or level n . But, we say that in our work out example, that when you need to compute the level 1, we only need level 0, then we can throw a level 0. When, we need to compute level 2, you need only level 1.

So, in some sense, you can keep only 2 copies and keep switching back and fold, you over write the zeroth level as second level, you over write the first level as third level and so on. So, we need only 2 slices of 1, call it of this three dimensional matrix at a time. So, you can just keep oscillating between these 2 slices overall here $2n^2$ space. So, we known normally worry about space.

But, just an observation that in this particular thing, you do not really need to have the n^3 array, you can have $2n^2$ array or a n^2 array with that two indices. And keep oscillating between the 2 and get the same effect, because you only need one copy to compute the other copy.

(Refer Slide Time: 14:20)

Historical remarks

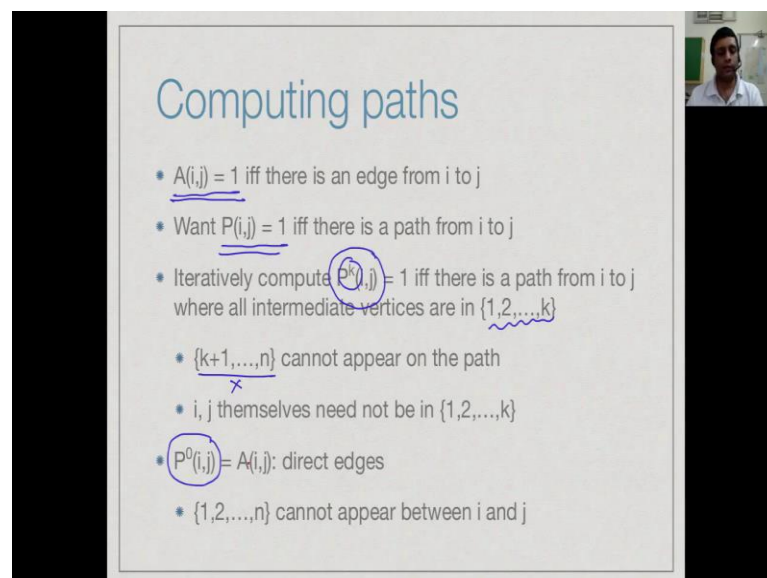
- Floyd-Warshall is a hybrid name *floyd* "Knows indirectly"
- Warshall originally proposed an algorithm for **transitive closure**
- Generating path matrix $P[i][j]$ from adjacency matrix $A[i][j]$
- Floyd adapted it to compute shortest paths

So, let us conclude this discussion in some historical remarks. So, Floyd Warshall as you can see come the hyphenation, as the hybrid name for this algorithm and actually there are two distinct algorithms which comprise it, which have a very similar structure. So, the original algorithm which are proposed by Warshall is for what is called transitive closure. So, transitive closure is exactly the same as computing the path from edge relation.

So, supposing you have a relationship like friend, so you know among a group of people, who is a direct friend of whom, then you might want to ask, who knows indirectly. So, I know somebody indirectly, if I have a friend, who knows that person or if I have a friend who as the friend knows that person and so on. So, knowing somebody indirectly is that transitive closure of the friend relation. In the same way in a general, I mean, so in every graph, if you put the friend, whatever relation, we want as the edge relation, the transitive closure is the path relation.

So, you have an adjacency matrix which represents the edges, you want to compute a path matrix which represents the paths, which are the pairs of vertices computed by which are connected by paths. And so, Warshall describe the similar algorithm, what we wrote now and we will just do it in a little detail in the next couple of slides to do this compute P from A and what Floyd is observed was you can adapted as same algorithm. So, if Warshall's algorithm, we only checking is there of path and Floyds algorithm says that, you can actually adapt it to compute shortest paths.

(Refer Slide Time: 15:50)



Computing paths

- $A(i,j) = 1$ iff there is an edge from i to j
- Want $P(i,j) = 1$ iff there is a path from i to j
- Iteratively compute $P^k(i,j) = 1$ iff there is a path from i to j where all intermediate vertices are in $\{1,2,\dots,k\}$
 - $\{k+1,\dots,n\}$ cannot appear on the path
 - i, j themselves need not be in $\{1,2,\dots,k\}$
- $P^0(i,j) = A(i,j)$: direct edges
- $\{1,2,\dots,n\}$ cannot appear between i and j

So, the idea is very similar to what you have seen for let us go through it quickly. So, we have an adjacency matrix A , which tells as the edges A_{ij} is 1, if there is an edge and we want a path matrix P P_{ij} is a 1, there is a path from i to j . So, we will again compute iteratively this quantity P^k_{ij} , it says that there is the path and this path uses only the vertices 1 to k .

So, k plus 1 to n cannot appear in the path and again the n points are not included. So, i

and j are arbitrary, it is what is a between i and j which is restricted by this super celebrity. So, between i and j , you can only see things from 1 to k . So, as before if you do P^0 , it says nothing can appear, because you could have everything from 1 to n , therefore, P^0 is just the adjacency matrix.

(Refer Slide Time: 16:43)

Inductively computing $P[i][j]$

- From $P^{k-1}(i,j)$ to $P^k(i,j)$ $\min(W^{k-1}(i,j), W^{k-1}(i,k) + W^{k-1}(k,j))$
- Case 1:** There is a path from i to j without using vertex k
 - $P^k(i,j) = P^{k-1}(i,j)$
- Case 2:** Path via $\{1, 2, \dots, k\}$ does go via k
 - k can appear only once along this path
 - Break up as paths i to k and k to j , each via $\{1, 2, \dots, k-1\}$
 - $P^k(i,j) = P^{k-1}(i,k)$ and $P^{k-1}(k,j)$
- Conclusion:** $P^k(i,j) = P^{k-1}(i,j)$ OR $(P^{k-1}(i,k)$ AND $P^{k-1}(k,j))$

So, now we have a very similar update rule, so if I know the paths which can be discovered using 1 to k minus 1, what are the paths I can discover using 1 to k . So, if there is a path already without using k , then I can just keep that path. So, we could either have that P^k , I should remember now this is there a path is not the path. So, this is like an adjacency matrix, there is no weight, it was the 0, 1 matrix or a true false matrix.

So, initially the adjacency matrix as 1 or true, whenever there is an edge, false whenever there is no edge. So, if I already got an entry true with k minus 1, then I can keep it. The other hand, maybe I do not have an entry true, after go back k , but one second that needs there is a path from i to k and there is a path from k to j and here I use only k minus 1 and here I use only k minus 1, because k needs to appear only once.

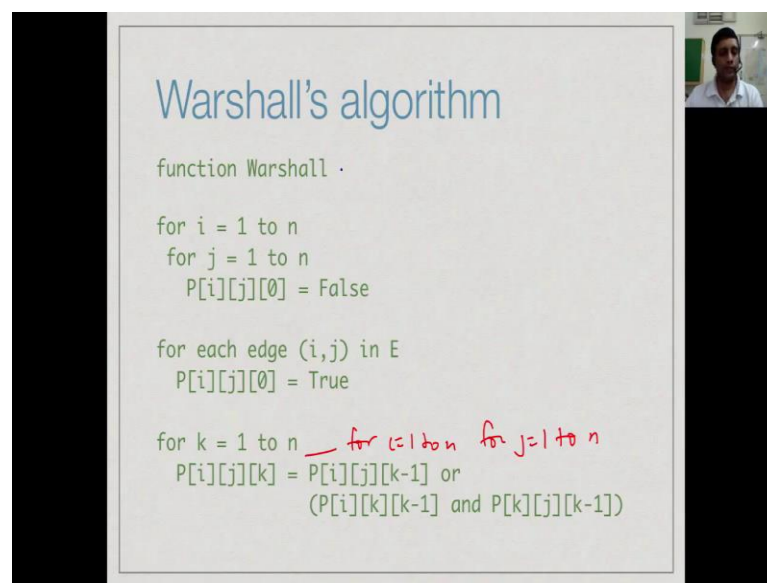
Just like in a shortest path, if I am just looking for some connectivity, then I do not gain anything by going back from k to k , because I can this remove that i and j will remain connected. So, I need to only look for paths which have one copy of every vertex along them. So, therefore, I can assume there is a path from i to k , which does not use anything outside want to k minus 1, likewise from k to j .

So, here now instead of min and plus operations, now become or and and, either I want

to path from i to k and path from k to j or I want to existing path from i to j , which never use k at all. So, I have this AND operation for combining these two existing paths should P and then, I have NOR operation which combines a case 1. So, earlier we had $W_{k-1, i j}$ and then, we add this $W_{k-1, i k}$ plus $W_{k-1, k j}$ and we took the min on this.

So, here instead of this plus, we are using AND and instead of using min, we are using OR, you can see that the algorithm that not exactly the same. But, it is very similar, so this was the algorithm proposed by Warshall.

(Refer Slide Time: 18:43)



Warshall's algorithm

```

function Warshall .
  for i = 1 to n
    for j = 1 to n
      P[i][j][0] = False

  for each edge (i,j) in E
    P[i][j][0] = True

  for k = 1 to n
    for i = 1 to n
      for j = 1 to n
        P[i][j][k] = P[i][j][k-1] or
          (P[i][k][k-1] and P[k][j][k-1])
  
```

So, you initialize everything to false and then, again you should P for i is equal to 1 to n for j is equal to 1 to n . So, you initialize all the paths to false, when you set explicitly that is the zeroth level path at true, if there is an edge. And then, you keep updating the k th level path the either saying that there was already $k-1$ level path or I can find 2 $k-1$ level paths were intermediate level k in intermediate vertex k .

So, it is a very similar thing, we just have just these operations of OR and AND, instead of min and plus. And then, Warshall's algorithm for use transitive closure and Floyd generalized it would shortest paths and these work in the presence of negative edges, it does not matter, Floyd's algorithm does not care with the edges are negative or positive. So, long as there are no negative cycles.