INDIAN INSTITUTE OF TECHNOLOGY MADRAS

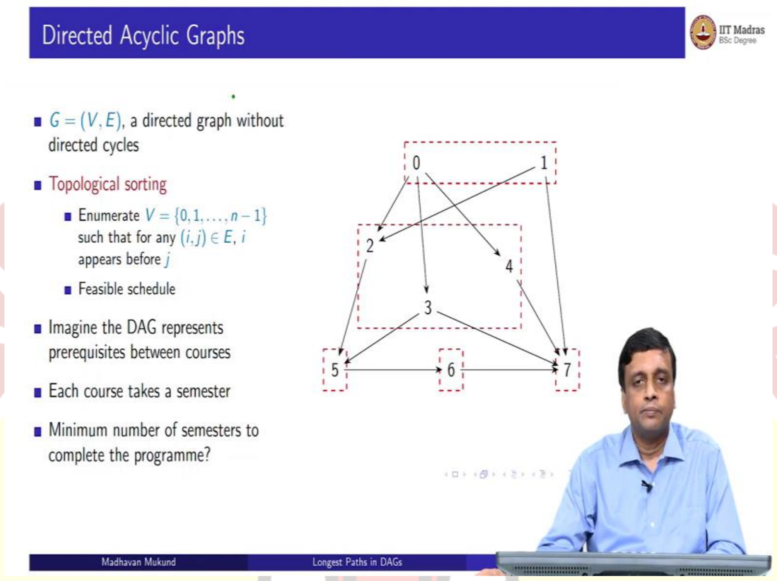
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Programming data Structures and Algorithms using Python
Professor Madhavan Mukund
Longest Path in DAGs

(Refer Slide Time: 00:13)



Directed Acyclic Graphs

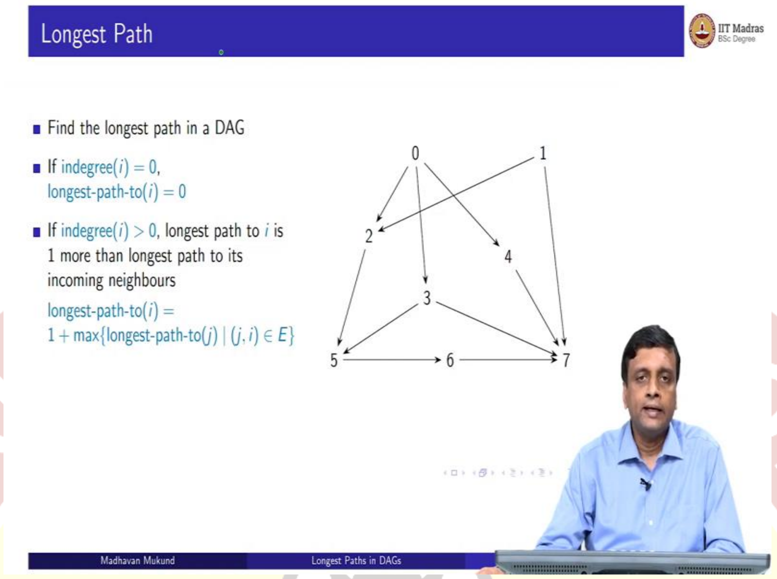
- $G = (V, E)$, a directed graph without directed cycles
- Topological sorting
  - Enumerate $V = \{0, 1, \dots, n-1\}$ such that for any $(i, j) \in E$, i appears before j
  - Feasible schedule
- Imagine the DAG represents prerequisites between courses
- Each course takes a semester
- Minimum number of semesters to complete the programme?

The second problem for DAGs is what is called the longest path. So, we have a DAG without cycles, we saw that a topological sort will enumerate this in some feasible order such that if i depend, if j depends on i if there is an edge from i to j, then I will always be enumerated before j. Now, let us look at a typical scenario. So, for example, supposing these are courses and these edges represent prerequisites, and it takes a semester to do a course, for example, then we can see how much time it will take for us to complete all the courses and finish the program.

So, each course takes a semester. And now we execute these courses or we take these courses as fast as possible. So, initially because 0 and 1 have no prerequisites, we can take them in the first semester, then having done 0 and 1, we find that we can do 2 and 4 because they only depend on 0 and 1. And we can do 3 as well. So, 2, 3, and 4 are courses we can take in the second semester. Now having done all of this, we still cannot take courses 6 and 7, because they depend on 5.

So, we have to do 5 alone in the next semester, having done 5 alone in the next semester, then we can do 6, and then we can do 7. So, this set of 0 to 7, 8 courses will actually take us 5 semesters to complete, given these dependencies.

(Refer Slide Time: 01:24)



Longest Path

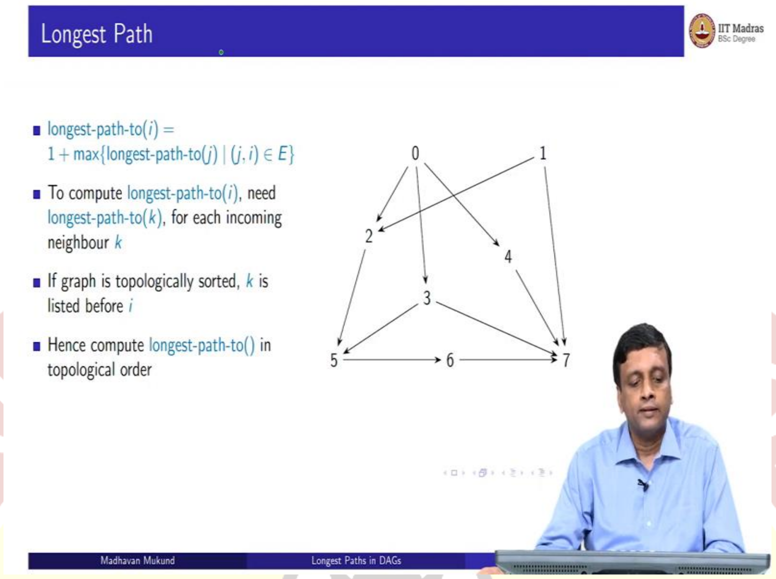
- Find the longest path in a DAG
- If indegree(i) = 0, longest-path-to(i) = 0
- If indegree(i) > 0, longest path to i is 1 more than longest path to its incoming neighbours

  longest-path-to(i) = $1 + \max\{\text{longest-path-to}(j) \mid (j,i) \in E\}$

So, our task now is to find the longest path in a DAG. And as you can imagine, this is closely related to the order in which they are enumerated. So, this is very closely related topological sort. So, if we have a DAG, and then the indegree of a node is 0, then the longest path to that node is 0. And that in other words, it can be done on the very first day, if you only think about it. So, there is no requirement the longest path represents how many requirements I need to satisfy sequentially in order to get to the task. So, if an indegree is 0, then I can do it immediately.

If it is not 0, then it has some incoming edges. So, if I knew how long it took to come there, the longest path to each of the incoming edges, then I can take the maximum among these, that is the constraint now I have to wait for all of them to be completed. So, I can take the one which is going to get completed latest, the maximum among the latest longest paths to all the incoming edges. And then my incoming long, my longest path will be 1 plus path

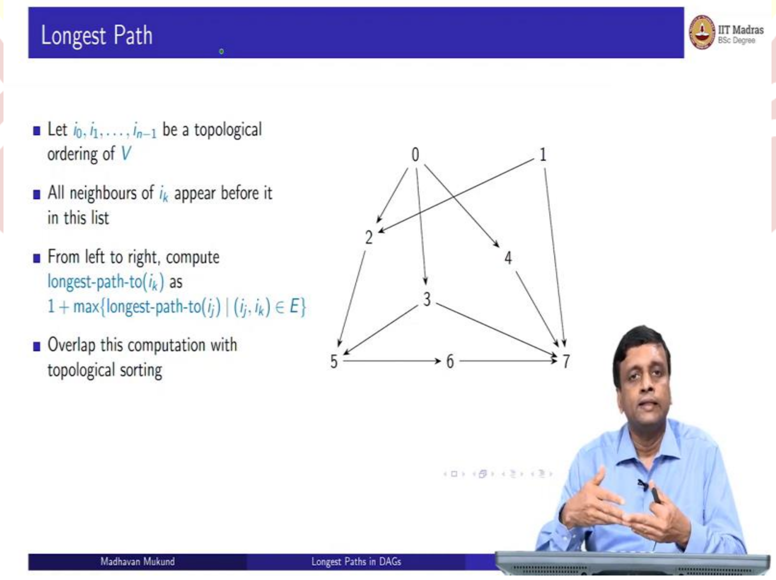
(Refer Slide Time: 02:30)


Longest Path

- longest-path-to(i) = $1 + \max\{$longest-path-to(j) $\mid (j,i) \in E\}$
- To compute longest-path-to(i), need longest-path-to(k), for each incoming neighbour k
- If graph is topologically sorted, k is listed before i
- Hence compute longest-path-to() in topological order

So, since I need to know the maximum of these longest paths, I need to enumerate them before I enumerate this vertex. But this is precisely the point of topological sorting. So, when I am processing a vertex k and trying to compute its longest path, I need to know the longest parts of everything which has an edge pointing into k, but in a topological ordering, this would have been done. So, if I, if I compute longest path following a topological ordering, then this kind of recursive or inductive definition that I have done here can be satisfied. So, what we will do is compute longest path in the topological order.

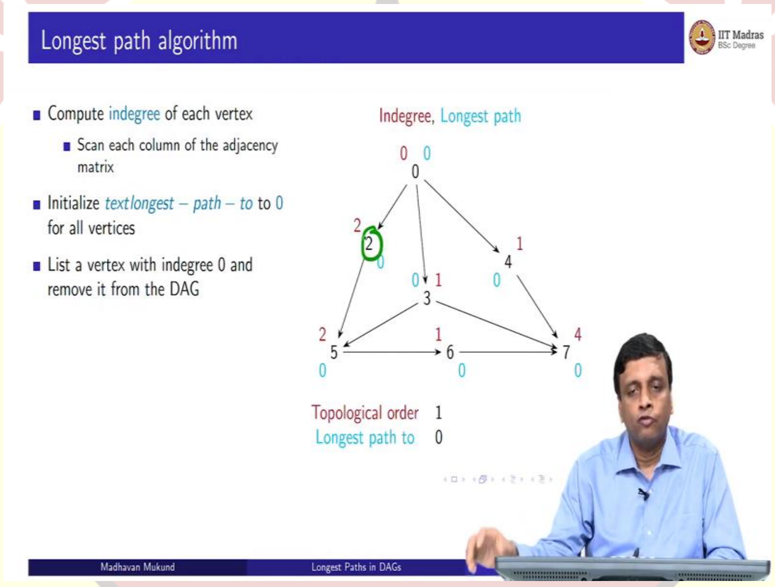
(Refer Slide Time: 03:09)


Longest Path

- Let $i_0, i_1, \ldots, i_{n-1}$ be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute longest-path-to(i_k) as $1 + \max\{$longest-path-to(i_j) $\mid (i_j, i_k) \in E\}$
- Overlap this computation with topological sorting

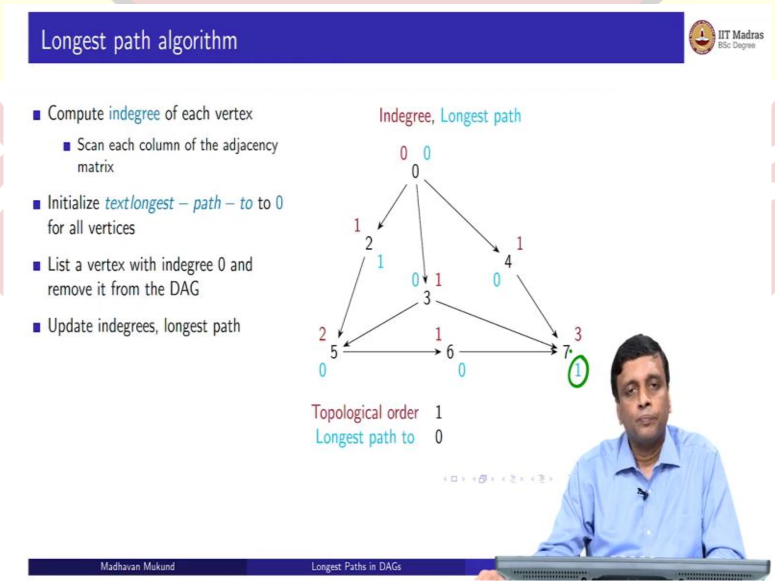
So, let us assume that we have some topological ordering of our vertices, then we know that everything in the every vertex in this list has all its neighbors appearing before it. So, we can go from left to right, having fixed a topological ordering, fixed to right, and we can do this. And we do not have to first compute the topological order and then scan it again as we are going along, as we are computing the topological order at any inductive point in a topological order, we have the same scenario that is everything that I depend on has been enumerated before me. So, I can inductively compute the longest path along with the topological ordering in one single pass.

(Refer Slide Time: 03:48)



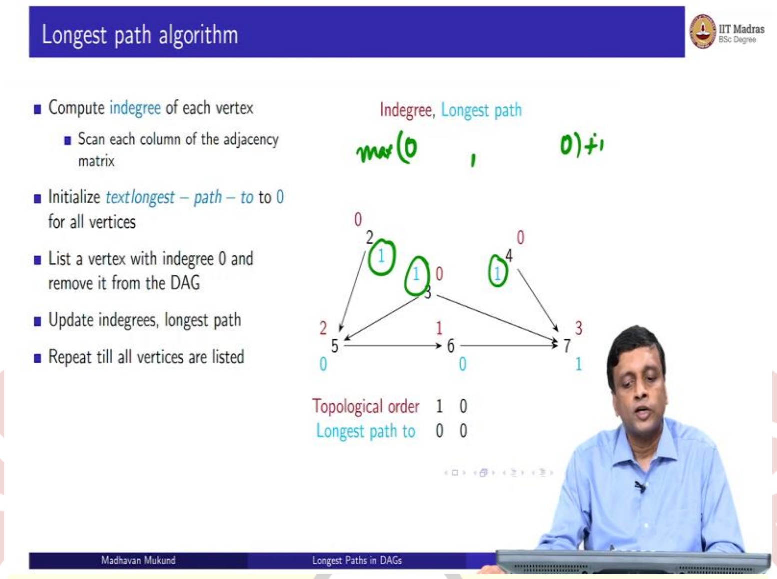
Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG



Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path



Longest path algorithm

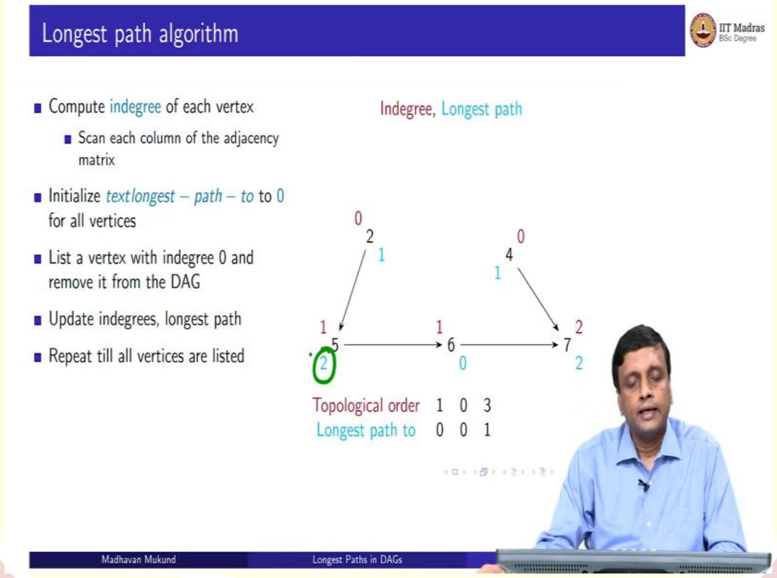
IIT Madras BSc Degree

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

max(0 , 0)+1

Topological order	1	0
Longest path to	0	0



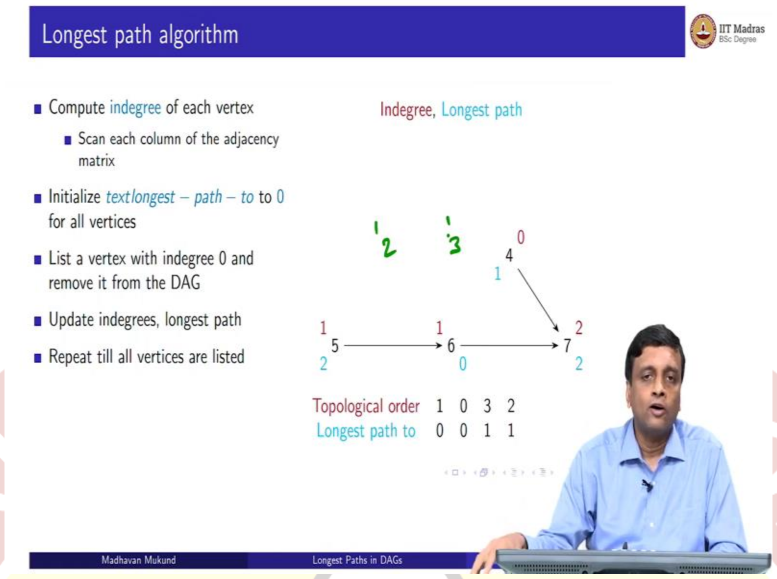
Longest path algorithm

IIT Madras BSc Degree

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

Topological order	1	0	3
Longest path to	0	0	1



Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

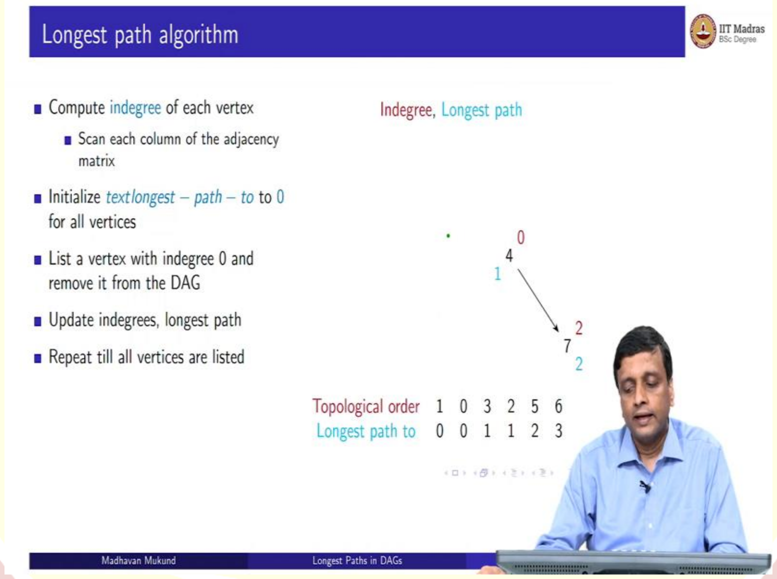
Indegree, Longest path

$$\begin{tikzcd} & & \overset{0}{\underset{1}{4}} \arrow[d] \\ \overset{1}{\underset{2}{5}} \arrow[r] & \overset{1}{\underset{0}{6}} \arrow[r] & \overset{2}{\underset{2}{7}} \end{tikzcd}$$

'2 '3

Topological order	1	0	3	2
Longest path to	0	0	1	1

Madhavan Mukund Longest Paths in DAGs



Longest path algorithm

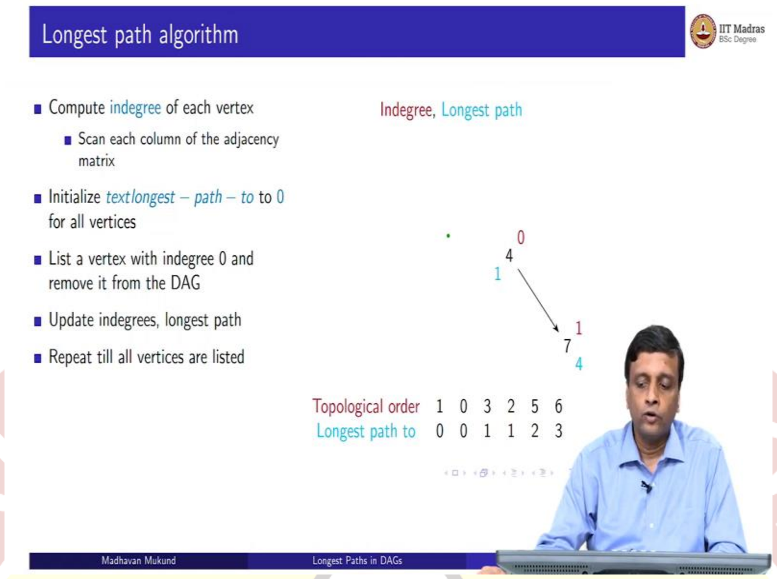
- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

$$\begin{tikzcd} \overset{0}{\underset{1}{4}} \arrow[d] \\ \overset{2}{\underset{2}{7}} \end{tikzcd}$$

Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3

Madhavan Mukund Longest Paths in DAGs



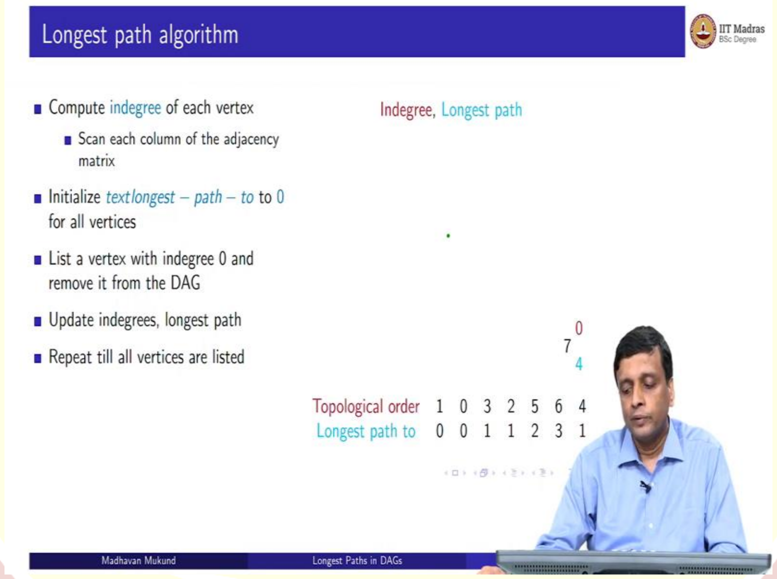
Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest* − *path* − *to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

$$\begin{tikzcd} \overset{0}{4}_{1} \arrow[dr] & \\ & \overset{1}{7}_{4} \end{tikzcd}$$

Topological order	1	0	3	2	5	6
Longest path to	0	0	1	1	2	3



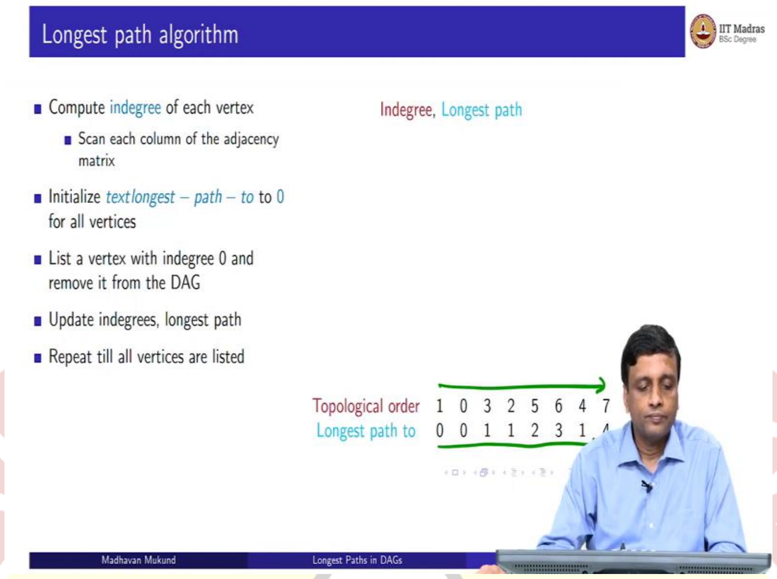
Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest* − *path* − *to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

$\overset{0}{7}_{4}$

Topological order	1	0	3	2	5	6	4
Longest path to	0	0	1	1	2	3	1


Longest path algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- Initialize *textlongest − path − to* to 0 for all vertices
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees, longest path
- Repeat till all vertices are listed

Indegree, Longest path

Topological order	1	0	3	2	5	6	4	7
Longest path to	0	0	1	1	2	3	1	4

Madhavan Mukund — Longest Paths in DAGs

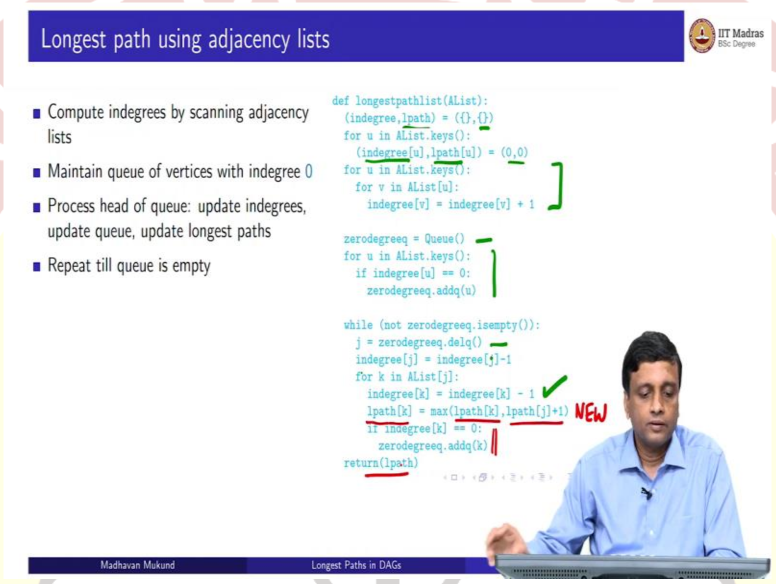
So, as before, we start by computing the indegree of every vertex, and now we will also simultaneously compute the longest path of every vertex. So, we initialize the longest path to be 0 by assumption. And now wherever I have indegree 0, it is 0. Now when I enumerate something, I eliminate it from the graph, I update the indegrees. But I will also update the longest path. So, now I know for instance, that it takes 2 takes me at least one step to reach 2 because I have to do something before it.

So, I will update the longest path to 2 and to 7 as being 1 plus the longest path to the vertex, which I just eliminated. So, I am incrementally updating the longest path. So, now if I enumerate 0, then again, I know that its longest path to 0, but it will now contribute 1 to both 3 and 4. Notice that the longest path to 2 remains 1, because the longest path to this was 0, the longest path to this was 0. So, it is the maximum of both these quantities plus 1, so I already knew it was 1. So, having enumerated the new vertex 0 does not give me any new information about the longest path to vertex 2.

Now I enumerate vertex 3. So, vertex 3 currently already has listed longest path of 1, and now 3 was pointing into vertex 5. So, now 5 must take 2 steps to be reached because it has to be reached via 3 and 3 already takes 1 step. So, the longest part of 5, which was earlier 0, as a default assumption has suddenly become 2. And I keep doing this. Next, I eliminate 2. And when I eliminate 2, nothing new happens because we had the situation like before, where 2 and 3 both had longest path 1. So, when I upgraded from 3, I already knew that 5 needed longest but 2, so I get no change.

So now, if I enumerate 5, then I get that 6 requires 3 steps, if I enumerate 6, then I get that 7 requires 4 steps. Finally, when I enumerate 4, I get no new information for 7 because 4 required only 1 step and 7 we already know requires 4 steps. So finally, enumerate 7 and I have this sequence. So, I have the sequence below, which I computed along with a topological sort above telling me the longest path to each of these vertices in my graph.

(Refer Slide Time: 06:19)



Longest path using adjacency lists

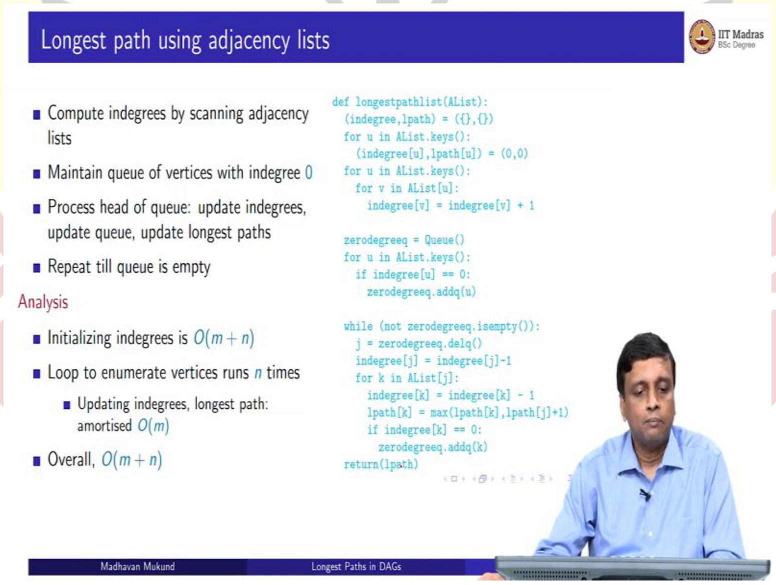
- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

```
def longestpathlist(AList):
  (indegree,lpath) = ({},{})
  for u in AList.keys():
    (indegree[u],lpath[u]) = (0,0)
  for u in AList.keys():
    for v in AList[u]:
      indegree[v] = indegree[v] + 1

  zerodegreeq = Queue()
  for u in AList.keys():
    if indegree[u] == 0:
      zerodegreeq.addq(u)

  while (not zerodegreeq.isempty()):
    j = zerodegreeq.delq()
    indegree[j] = indegree[j]-1
    for k in AList[j]:
      indegree[k] = indegree[k] - 1
      lpath[k] = max(lpath[k],lpath[j]+1)   NEW
      if indegree[k] == 0:
        zerodegreeq.addq(k)
  return(lpath)
```

Madhavan Mukund Longest Paths in DAGs



Longest path using adjacency lists

- Compute indegrees by scanning adjacency lists
- Maintain queue of vertices with indegree 0
- Process head of queue: update indegrees, update queue, update longest paths
- Repeat till queue is empty

Analysis

- Initializing indegrees is $O(m + n)$
- Loop to enumerate vertices runs n times
  - Updating indegrees, longest path: amortised $O(m)$
- Overall, $O(m + n)$

```
def longestpathlist(AList):
  (indegree,lpath) = ({},{})
  for u in AList.keys():
    (indegree[u],lpath[u]) = (0,0)
  for u in AList.keys():
    for v in AList[u]:
      indegree[v] = indegree[v] + 1

  zerodegreeq = Queue()
  for u in AList.keys():
    if indegree[u] == 0:
      zerodegreeq.addq(u)

  while (not zerodegreeq.isempty()):
    j = zerodegreeq.delq()
    indegree[j] = indegree[j]-1
    for k in AList[j]:
      indegree[k] = indegree[k] - 1
      lpath[k] = max(lpath[k],lpath[j]+1)
      if indegree[k] == 0:
        zerodegreeq.addq(k)
  return(lpath)
```

Madhavan Mukund Longest Paths in DAGs

So, here is a variation of our topological sort algorithm, which we are now directly giving in terms of an adjacency list, because we saw that for a topological sort, the list makes a lot more sense in adjacency matrix. So, the new thing here is that we are keeping track of this longest path metric. So, we are not really interested. So, we are implicitly doing the topological sort. But the purpose of this particular function is not to give us a topological order.

So, we are not keeping track of what we were doing earlier, which was the topologically sorted list. Instead, we will keep this longest path function as a dictionary, longest path l path of I will be the longest path to i. So, we initialize the in degree and the longest path of every vertex to be 0. Now, as before, we will walk through all the vertices and update the degree in time proportional to the number of vertices.

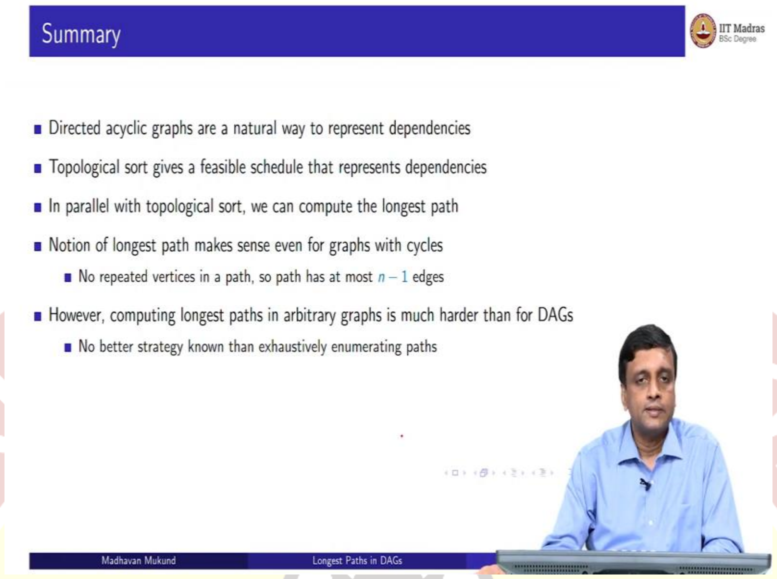
Now as before, because we are doing the same topological calculation sort calculations before we keep the 0-degree queue, and we put all the 0-degree vertices into the queue. So far, we have done nothing different from what we are doing before, except that instead of keeping track of this topological sort lists, we are keeping track of this l path longest path dictionary.

So now, as long as there are 0-degree vertices to process, we take out the highest one. So now for every outgoing edge from here, for every k, which is in the adjacency list of j, we update, its indegree as before, and now is the new step. So, this is new. So, what we are doing now is we are saying the longest path of k is the maximum of what we already knew, and 1 plus the longest part of j, which we have just discovered.

So, this is the only difference actually between the topological sort algorithm and the longest path algorithm, except for the fact that we are, we can also keep track of the topological sort here if we wished, but we have not done so. And then, as before, if we discovered that the newly decremented degree has become 0, we put it into the queue so that it will be processed when its turn comes.

And finally, instead of returning the topological sort list, we are returning this path, this longest path dictionary. So, this is basically very similar to topological sorting and so is the analysis, initializing takes n plus n time, and inside the loop, which runs n times, we do an update which is an amortized m time update, because we have to do it to the sum of the degrees and so, this is m plus n.

(Refer Slide Time: 08:53)



Summary

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
- In parallel with topological sort, we can compute the longest path
- Notion of longest path makes sense even for graphs with cycles
  - No repeated vertices in a path, so path has at most $n-1$ edges
- However, computing longest paths in arbitrary graphs is much harder than for DAGs
  - No better strategy known than exhaustively enumerating paths

So, basically with directed acyclic graphs now, we can do both these things very efficiently we can get the feasible schedule with topological sort. And while we are doing the topological sort, we can also compute through the dependencies the longest path that is the minimum time. So, the longest path is really telling me the minimum time it will take me to finish all the tasks according to the dependencies.

Now, of course, the longest path makes sense for any graph and I can take any graph which does not have cycles and say if I have cycles, then the way we have defined path, we can go round and round. So, we can basically look at paths which do not repeat vertices and ask what is the longest path in a graph, so this is often called the diameter of the graph.

Now, it turns out that computing this in general is very hard. So, when DAGs we have a very efficient algorithm, we have something which is essentially linear in the size of the DAG, remember o m plus n is basically linear in the size of the graph. And if we go to a non-DAG then this problem becomes hopelessly complicated. So, short of trying out all paths and finding out which is the longest there is no really good strategy which works across all graphs. So, longest path in general is a very hard problem in graphs, but for DAGs it is surprisingly simple.