



IIT Madras
ONLINE DEGREE

Mathematics for Data Science 1
Professor. Madhavan Mukund
Department of Computer Science
Indian Institute of Technology, Madras
Lecture No. 68
Topological Sorting


(Refer Slide Time: 00:14)

00:18 / 20:10

Topological Sorting

Madhavan Mukund
<https://www.cmi.ac.in/~madhavan>

Mathematics for Data Science 1
Week 11

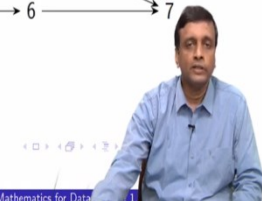
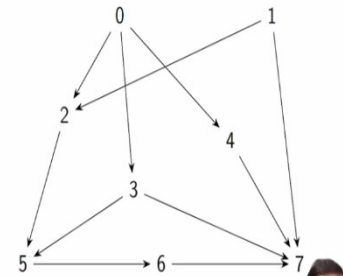


So, we have motivated the use of DAGs by saying that they are useful for representing tasks and dependencies.

(Refer Slide Time: 00:21)

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
- Represents a feasible schedule



Madhavan MukundTopological SortingMathematics for Data Science 1

And 1 of the things that we need to do in a DAGS in a directed acyclic graph is to arrange the vertices in a list that respects the dependencies. So, we said this is a Topological sort. In a Topological sort of the vertices, we sequentially list out the vertices in such a way that every time there is a directed edge from i to j , i must appear before j in the sequence that we list out.

So, in terms of the applications that we are thinking of, if we think of these as tasks and dependencies, then this type of a topological sort represents a feasible schedule. A schedule in which no task appears before all the tasks it depends on are already completed. So, when you come to do something, everything you need to do before that is already done.

(Refer Slide Time: 01:04)

Topological Sort

- A graph with directed cycles cannot be sorted topologically
- Path $i \rightsquigarrow j$ means i must be listed before j
- Cycle \Rightarrow vertices i, j such that there are paths $i \rightsquigarrow j$ and $j \rightsquigarrow i$
- i must appear before j , and j must appear before i , impossible!

Claim
Every DAG can be topologically sorted

Madhavan Mukund Topological Sorting Mathematics for Data Science

So, the first thing to notice is that if you have cycles, then you cannot do this. If you have a graph with directed cycles, we informally argued that if i depend on somebody else to finish, and somebody else depends on me to finish, then we cannot do anything because we are waiting for each other to finish. But formally, let us see what it means. So, in the topological sort, it is not just for every edge, for every edge i comma j , it is clear that topological sort requires i to appear before j .

But in general, if I have a path of dependencies, if i depends on k and k depends on j , then i must come before k and k must come before j . So therefore, transitively i must come before j . So, anytime there must there is a path from i to j it must be listed before j . So now, if I have a cycle, in a directed graph, it means that I can go from some i to some other j . So remember, a cycle cannot

be just something which goes from i to i without going to any other vertex, we need to go at least cross 1 edge. So, I must go from i to some other vertex j and then come back. So, there is a path from i to j and a path back from j to i .

Now, by the previous requirement, if there is a path from i to j , then the topological sort is obliged to put i before j . But since there is also a path from j to i , then we have to put j before i . But clearly in a given sequence either i can come before j or j can come before i we cannot have both these constraints satisfied in the sequence, and therefore this would be impossible. So, that is why if we have a cycle of dependencies, there is no way to order them in a feasible sequence, such that each task appears only after everything it depends on has happened before.

So, what we are going to do now is to show the other side of this, so what we said is if it is not a directed acyclic graph, so it is directed, but with cycles, then topological sorting is always impossible. On the other hand, what we need to argue is that if you give me a feasible set of constraints, in the sense that there are no cycles, there are no cyclic dependencies, it is a DAG, then I will always be able to complete it in some reasonable order. So, every DAG can be topologically sorted.

(Refer Slide Time: 03:14)

How to topologically sort a DAG?

Strategy

- First list vertices with no dependencies
- As we proceed, list vertices whose dependencies have already been listed
- ...

Questions

- Why will there be a starting vertex with no dependencies?
- How do we guarantee we can keep progressing with the listing?

Madras Institute of Technology

So, how would we go about doing this? So, clearly I have to begin somewhere, so the first thing I have to do is list out a task, which has no dependencies, it does not require anything else to be done. So, there must be a vertex with no dependencies, there may be more than 1. If you look at

this graph on the right, we see that 0 has no incoming edges. So, 0 does not depend on anything. 1 has no incoming edges. So, I could start with 0 or I could start with 1. But in general, I need to find such a vertex which has no incoming dependencies to start with.

As I complete the dependencies, a later vertex with depends on a few things now becomes available, because everything that needed to be done before that is done. So, as long as we can find vertices whose dependencies have already been listed, we can then list these. So, this is our general strategy. So, we first start with something which has no dependencies. And every time we have a dependency satisfied, we strike it off the list. So, then vertices, which do have dependencies, eventually, all the dependencies have already been listed, and then I can list them.

So, in order to apply the strategy, we need to of course, guarantee that there will be a starting vertex with no dependencies. Otherwise, there is no way we can start. And then we also have to guarantee that eventually, every vertex which has dependencies will find all those dependencies listed out. So, we have to make sure that every vertex, remember that we said that whenever we have a DAG, we can do a topological sort, this is our claim.

And this strategy above says that we are going to do this by starting with a vertex with no dependencies. So, first we have to show that such a vertex exists. And then we have to argue that as we progress, we are definitely going to eliminate all the vertices in our list and finally finished by doing all the tasks or listing them all out in this topological sort. So, we need to somehow justify these claims in order to proceed with this strategy.

(Refer Slide Time: 05:09)

Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim
Every DAG has a vertex with indegree 0

- Start with any vertex with $\text{indegree} > 0$
- Follow edge back to one of its predecessors
- Repeat so long as $\text{indegree} > 0$

Maths for Data Science

So, remember that in a directed graph, we talk about the indegree and the out degree of a vertex as opposed to just the degree. So, in an undirected graph, the degree of a vertex refers to the number of edges which are incident on that vertex, how many edges have that vertex as an endpoint. But in a directed graph, these edges could either come in, or they could go out. So, the indegree is how many edges are pointing into a vertex v .

What we are looking for is a vertex with no dependencies, that means nothing is pointing into it v does not depend on anything, so there is no edge of the form $u \text{ comma } v$. So, why must there be such a vertex? So, the claim is that every time must have such a vertex within degrees 0. So, let us suppose we have vertices within degree not 0, so pick any 1. So, we start with the vertex v , which has indegree greater than 0. Since it has indegree greater than 0, there is at least 1 edge coming into that vertex. So we can follow that edge backwards and go to a preceding vertex. So, we can go back from v from a preceding vertex.

So, let us say supposing we start here, we say that this vertex, has some incoming edge. So, I go across this vertex, I go backwards and I come to this vertex. Now I say this vertex also has a nonzero number of incoming things it has integrated in 0. So, I will keep doing that. So, I will start here, then I will go back here and then maybe I will go back here. And then maybe I will go back here. And then I stop, because I cannot go back any further.

So, in this particular case, this graph is acyclic and I have stopped at 0 by starting at 6. If I started at 7, I could follow a different path. For example, I could go from 7, I could go to say, 4 and then 0, or from 6, I could have gone from 5 to 2 to 1 and so on. But whichever way I do it, eventually all these paths will have to stop. And why is this the case?

(Refer Slide Time: 07:11)

Algorithm for topological sort

- A vertex with no dependencies has no incoming edges, $\text{indegree}(v) = 0$

Claim
Every DAG has a vertex with indegree 0

- Start with any vertex with $\text{indegree} > 0$
- Follow edge back to one of its predecessors
- Repeat so long as $\text{indegree} > 0$
- If we repeat n times, we must have a cycle, which is impossible in a DAG

Well, supposing I started some v , let me call it v_1 , or v_0 , if you want it, then I come back to another vertex, which is v_1 . And then I cannot stop because v_1 has indegree greater than 0, so I have to go back to another vertex which is v_2 . Now, if I hit the same vertex again, then I have a cycle. So, let me assume that v_1 is different from v_0 and v_2 is different from v_1 and v_0 . So I am continuously hitting new vertices as they go along. If I do not hit a new vertex, every time I go backwards, I have already found a cycle and I know this graph has no cycles, but on the other hand, this graph has only n vertices.

So, if I started v_0 and I do 1 step, I get back to v_1 , if I do 2 steps, I get to v_2 . So, if I do $n - 1$ steps, I have reached (v_{n-1}) . So, after I have done this, if I do it one more time, if I do an n th step backwards, then I cannot find a new vertex anymore, which I have not seen before, because all the n vertices in my graph have already been traversed somewhere in that path that I have seen so far. So therefore, the new vertex must be going back somewhere here. So, it must be one of the vertices already seen. So, there must be a cycle.

So, there is a directed cycle. And since it cannot be a directed cycle, this cannot happen. So, this is a complicated way of proving something is called proof by contradiction. So, you say assume that everything has a nonzero indegree, then I can find a path, which is arbitrary length, in particular of length n , which will visit $n + 1$ vertices. And since $n + 1$ vertices must repeat a vertex, there must be a cycle and this cannot happen. So, this is why we will always have a starting point, we will always have a starting point, which is an indegree vertex with a vertex with indegree 0.

(Refer Slide Time: 08:56)

Topological sort algorithm

Fact
Every DAG has a vertex with indegree 0

- List out a vertex j with $\text{indegree} = 0$
- Delete j and all edges from j
- What remains is again a DAG!
- Can find another vertex with $\text{indegree} = 0$ to list and eliminate
- Repeat till all vertices are listed

So, this claim is now a fact. So, that was a proof that we have a vertex which is guaranteed to have indegree 0. So, in this particular graph, as we said, we have the vertices labeled 0 and 1, which both have no edges pointing into them. Now, what do we do? Well, we list it out, because we start from there. And once we list it out, we kind of pretend that it is no longer a constraint because it has been done.

So, if it was a constraint for somebody, for example, if we look at vertex 4, for instance, so vertex 4 requires vertex 0 to be completed, but if I list out 0 saying 0 is done, now, 4 has no longer any constraints, because this constraint is gone. So, this was a claim that as we go along, the constraints will go away until you can list out the later vertices. So, if I delete that vertex that I just found, and all the edges from j , what happens?

So, if I delete this, and then I delete all the edges that point out of 0, then I am left with a smaller graph in which there is 1 less vertex, and a few small, fewer edges depending on how many edges I had connected to that original vertex. But notice that in this process, I have only removed edges from a directed graph. And if the original graph had no cycles, this must also have no cycles, because I have not put back any edge between 2 vertices which are not already connected. So, what remains after this is again a DAG. So, I take a DAG, I remove any vertex from it, it remains a DAG it may not be connected for if I have done it badly, but at least it cannot have any cycles. So, it will be directed and it will be acyclic.

And therefore, by the same argument in the new DAG, that is, after I have believed this constraint has been satisfied, I must again have some vertex with indegree 0. So, at every stage, I have a DAG, whenever I have a DAG, by that earlier argument, there must be a vertex with indegree 0. So, at every stage, there are at least 1 vertex which I can remove. And I keep doing this and after n minus after n stages, I must have removed all the vertices. So, this is how the procedure works.

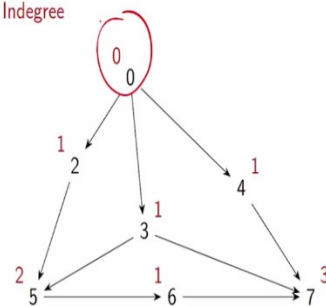
So, we repeat this process until all the vertices are listed. And we are guaranteed that at every stage, start with a DAG, remove a vertex of degree 0, I have another DAG, therefore, I have another vertex of indegree 0, remove that I have 1 more, and so on. So, that is why this process is guaranteed to make progress, and is guaranteed to exhaust all the vertices in my DAG.

(Refer Slide Time: 11:24)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees

Indegree



Topologically sorted sequence

1,



So, the first step to implement this as an algorithm is to compute the indegree. So, assume that we have the graph presented to us as usual as an adjacency matrix, then we know that the incoming edges are in the columns. So, remember that the row i has all the outgoing edges from i , and the column i has every entry of the form j comma i , which is edge from j to i . So, if I look at the column i it has all the entries for the incoming edges.

So, if I just walk down column i and add up the 1s, remember that this matrix has 0 1 entries, if I just add up the 1s, I will get the indegree. So here, we have got this graph and by doing this one scan, although we can do it pictorially in this particular case, by doing this one scan, we can count the incoming arrows. So for instance, this has degree indegree 2, because there are 2 edges coming in this as indegree 4, because there are four edges coming in. But this is not something we have to do with the picture, we can actually just mechanically do it using the adjacency matrix.

So, now that we have this, now we can compute. So, we have an alternative, a second list, in some sense a list of indegrees of every vertex, we are, we know in advance that there must be at least 1 of these vertices which has indegree 0, we do not know which 1, but we can find it by just scanning down the list and looking for a 0. So, we go down the list and we look for a 0 and perhaps we decide to pick so there are 2 in this case, as we know.

So, we have both what is 0 and 1. So, let us suppose we choose to do this one. So remember, our procedure says list it out and remove it from the graph. So, we list it out here to the bottom is our list, and we remove it from the graph. So, the edges, which are now pointing out of vertex 1 have been removed. And in this process, the targets of those edges indegree has reduced. So, when I do this, I had I had an edge I claim like this.

So, this indegree and this indegree now will change. So, when I remove that vertex, I must also simultaneously update the indegrees, I do not have to scan all the indegrees. Again, I only have to look at the row for i the vertex i just deleted as i , I only look at the row for i and every ij that I have as an edge, I look at the degree of j and I reduce it by 1. So in this case, I had 1 to 2 and 1 to 7. So I go to vertex 2 and reduces indegree by 1, I go to vertex 7 and reduces indegree by 1. Now again, I have a DAG a smaller dag, again, I must have a vertex, at least 1 of indegree 0 here, I have no choice, I have only this one. So, I remove that and list it out. And now again, these are the three vertices which were getting edges from the vertex 0, which has this just deleted.

(Refer Slide Time: 14:10)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

Topologically sorted sequence
1, 0, 3,

Madhavan Mukund Topological Sorting Mathematics for Data Sci

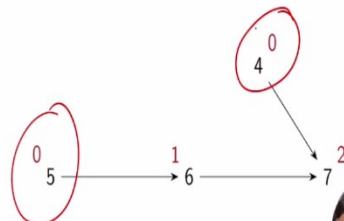
So, now I have to reduce they are indegrees by 1. Now I have a wide variety of vertices with indegree 0, which I can enumerate next, because all of them depended on 1 and 2. So, I pick any 1 of them. So, for instance, I picked 3, the one in the middle, I picked 3 and I remove it, so I list it out. And now 3 was pointing in this direction and this direction, so this 2 will have to reduce and this 3 will have to reduce. So, I reduce them. Now again, I have 2 choices, 2 and 4. So, perhaps I do 2 next.

(Refer Slide Time: 14:42)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



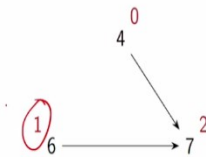
Topologically sorted sequence

1, 0, 3, 2,

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



Topologically sorted sequence

1, 0, 3, 2, 5,

So, I take 2, and now I reduce the indegree of 5 by 1. Now I have these 2 candidates to enumerate next, so perhaps I seek vertex 5. And then I have to reduce the indegree of 6 by 1. And now maybe I do 6 next.

(Refer Slide Time: 15:02)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



Topologically sorted sequence

1, 0, 3, 2, 5, 6,

Madhavan Mukund

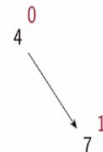
Topological Sorting

Mathematics for Data

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree



Topologically sorted sequence

1, 0, 3, 2, 5, 6,

Madhavan Mukund

Topological Sorting

Mathematics for Data

सिद्धिर्भवति कर्मजा

Topological sort algorithm



- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

1
7

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,



Topological sort algorithm



- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Indegree

0
7

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4,



INDIAN INSTITUTE

TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed

Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4, 7

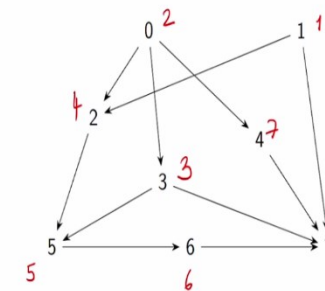


And then I reduce the indegree of 7 by 1. But I still cannot enumerate 7 because it has indegree 1, but 4 is left with indegree 0. So, I can enumerate for next, reduce the indegree of 7 to 0. And finally, I can enumerate 7.

(Refer Slide Time: 15:22)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed



Topologically sorted sequence

1, 0, 3, 2, 5, 6, 4, 7



So, if we look at our original graph, this is what the graph looked like. So, what we have said is that we did this first, then this, then this, then this, then this, and this, then this. And then. So, this is the sequence in which we enumerated it, perhaps not the most obvious sequence that you would have thought of, we might have thought of doing it top to down to 01, and then maybe 234, and then 567. But this is a valid sequence.

(Refer Slide Time: 15:52)

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed
- Using adjacency lists?
 - Scan each list $i \rightarrow [j_1, j_2, \dots, j_k]$
 - Increment $\text{indegree}(j_i)$ for each j_i

Topologically sorted sequence: 1, 0, 3, 2, 5, 6, 4, 7

Handwritten notes: $0 \rightarrow [2, 3, 4]$, $1 \rightarrow [7]$, $\text{indegree}(2) + 1$, $(3) + 1$, $(6) + 1$

So, what if we had adjacency lists instead of adjacency matrix, so we said did an adjacency matrix representation, we can find the incoming edges by looking at the relevant column, we look at column i , and we have done it. In adjacency list, we only have outgoing edges. If I look at the list for i , it has only edges pointing out of i . So, how do I get the edges pointing into i . Well, you do not do it in one shot.

So, for the column thing, you compute the indegree of i in one shot by looking at the column for i , here you look across all the lists, so you will start with vertex 0, and look at the list of things that 0 is pointing to. So, 0 is pointing to, in this case, 2, 3, and 4. So, what you will do is you will have separately indegree of 2, indegree of 3 and indegree of 4. So, when I see this 2, I will do a + 1 here, when I see this 3, I will do a + 1 here, when I see this 4, I will do a + 1 here.

Now I come to 1, so 1 has outgoing 2 and 7. So, now I will do another + 1 here, and in 7, I will do a + 1, and so on. So, you basically scan all the lists from top to bottom. And for each outgoing as you see, you go to the corresponding target indegree and incremented by 1. So, even if you have an adjacency list, we can do a simple scan, and update all the indegree to start with. After that it is only a matter of checking the indegree as you are deleting them. So, it does not matter whether it is incoming or outgoing.


So, we have the usual caveats as we had before, that is if you are doing an adjacency matrix, everything takes order n time because you have to look at all the outgoing edges or all the things

but as we are doing something, which is the adjacency list, you can do it in time proportional to the total number of edges.

(Refer Slide Time: 17:20)

Summary

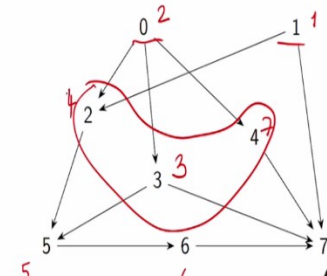
- Directed acyclic graphs are a natural way to represent dependencies
- Topological sort gives a feasible schedule that represents dependencies
 - At least one vertex with no dependencies, indegree 0
 - Eliminating such a vertex retains DAG structure
 - Repeat the process till all vertices are listed
- More than one topological sort is possible
 - Choice of which vertex with indegree 0 to list next



N indeg tasks n! orderings

Topological sort algorithm

- Compute **indegree** of each vertex
 - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with **indegree = 0** to list and eliminate
- Repeat till all vertices are listed



Topologically sorted sequence
1, 0, 3, 2, 5, 6, 4, 7

So to summarize, we have seen from the earlier lecture, that directed acyclic graphs are a very natural way to represent dependencies. And one of the fundamental problems in such a situation is to find a feasible schedule, find a sequence in which you can perform the tasks or do whatever we need to do, which does not violate any of the constraints. So, something must be listed before something that that depends on it.

So, what we observed is that in any DAG, there has to be at least one vertex which has no dependencies, there is something within the graph version on the directed acyclic graph representing the dependencies has indegree 0, so we can list it because it has nothing that has to come before it. And eliminating a vertex from a DAG gives us back another DAG smaller DAG, possibly disconnected.

So, we could have a DAG for instance, we look like this, this is a DAG. So, it says that I must do this, say this is my first task. And I must do this before 2 and 3. Now once I have done this, now I have a DAG, which consists of just 2 and 3. So, we might end up with a disconnected graph, but it is still without cycles. And therefore, by the same logic, it must have something of indegree 0 and so we can keep repeating and that is why this process works.

Now, the other thing to notice is that more than one topological sort is possible. So we saw that when we looked at that example of how to set up the room, we said that for instance, tiling the floor and plastering the walls can be done in either order. And in particular, what happens is that when we end up with multiple choices for indegree 0 vertices.

So, if we look at our previous example, for instance, I could have started with 0 or with 1 we chose to start with 1 if we started with 0 we would have got a slightly different sequence. Similarly, we had a situation when we had 2 3 and 4 all available to us with indegree 0. So, we chose to do 2 first we could have done 3 first, we could have done 2 first we could have done 4 first.

So, whenever we have multiple degree vertices with indegree 0 topological sort does not necessarily force us to take one or the other we might choose to take the smallest one in which case we get one particular order, but there are multiple orderings possible. So, this is a thing that we need to remember that Topological sort produces a sequence which is compatible, but this is by no means the only sequence there are multiple topological orderings possible.

In particular, if you have no dependencies if I have all the tasks are independent, that any ordering is possible. So, if I have this is basically if I have n independent tasks, then I would have $n!$ orderings. So, the number of topological orderings can be very large so we are not really interested in computing at this point the number of topological ordering we are also interested in finding one of them.