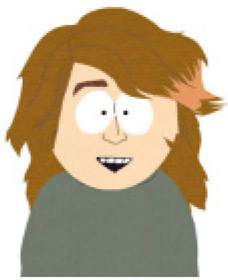
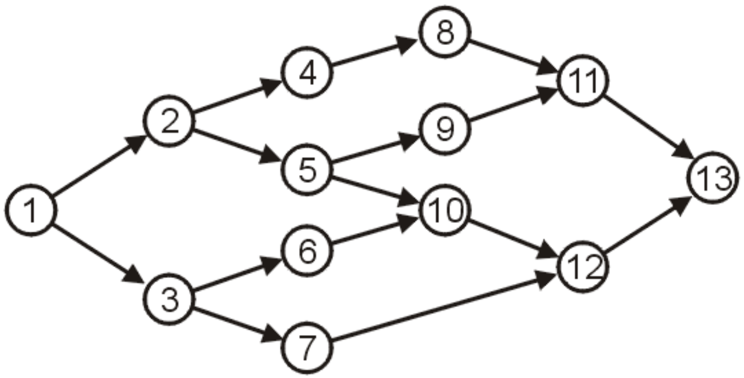
Topological sort





Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Topological sort

Standard reminder to set phones to silent/vibrate mode, please!

Topological sort

- During today's class, we will:
  - Look at Topological sort, a common and useful operation with Directed Acyclic Graphs (DAGs)
  - Discuss an algorithm to implement this operation.
  - Briefly talk about some of its applications.

Topological sort

- The basic idea of a topological sort is the following:
  - Given a DAG, in which we could see adjacencies as representing a pre-requisite task, we want to place the vertices in sequence.
  - The sequence must be one in which all dependencies on pre-requisites are satisfied.
  - Such a sequential arrangement of the vertices is called a *topological sort* of the DAG.

Topological sort

- A simple and obvious application is:
  - We have a DAG where vertices are tasks that we want to perform (e.g., part of a compiler's code generation subsystem).
  - We need to execute all the tasks, but we can only do one at a time.
  - A topological sort gives a valid sequence of executed tasks — no task can proceed until all pre-requisite tasks have completed.

Topological sort

- Formally, we would define a topological sort as follows:
  - Let $G = (V, E)$ be a DAG, with $V = \{v_1, v_2, \cdots, v_{n-1}, v_n\}$

Topological sort

- Formally, we would define a topological sort as follows:
  - Let $G = (V, E)$ be a DAG, with $V = \{v_1, v_2, \cdots, v_{n-1}, v_n\}$
  - Then, a topological sort is a sequence containing all the elements in V, $\{v_{k_1}, v_{k_2}, \cdots, v_{k_{n-1}} v_{k_n}\}$ such that for all i, j ($0 \leqslant i, j \leqslant n$), if there exists a path from v_{k_i} to v_{k_j}, then $i < j$.

Topological sort

- Formally, we would define a topological sort as follows:
  - Let $G = (V, E)$ be a DAG, with $V = \{v_1, v_2, \cdots, v_{n-1}, v_n\}$
  - Then, a topological sort is a sequence containing all the elements in V, $\{v_{k_1}, v_{k_2}, \cdots, v_{k_{n-1}} v_{k_n}\}$ such that for all i, j ($0 \leqslant i, j \leqslant n$), if there exists a path from v_{k_i} to v_{k_j}, then $i < j$.
    - Simply put, if there is a path from vertex v to vertex w, then v appears before w in the output sequence.

Topological sort

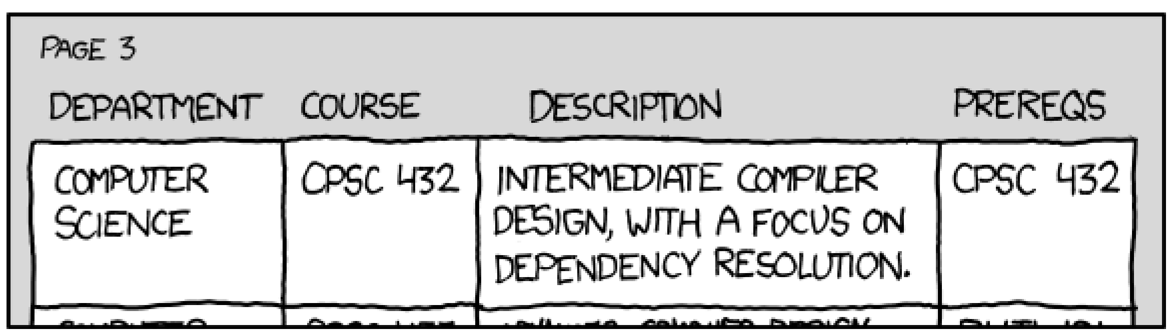
- Question: why is the notion specific to DAGs?

Topological sort

- Question: why is the notion specific to DAGs?
- Let's have xkcd answer that question for us, pointing out that cycles in dependencies can be problematic:

Topological sort

- Question: why is the notion specific to DAGs?
- Let's have xkcd answer that question for us, pointing out that cycles in dependencies can be problematic:



http://xkcd.com/754/

Topological sort

- In fact, let's look at (and prove) this interesting fact:
  - A directed graph is a DAG if and only if it has a topological sort.

Topological sort

- In fact, let's look at (and prove) this interesting fact:
  - A directed graph is a DAG if and only if it has a topological sort.
  - Proof:

    We observe that there are two independent statements to prove:
    - A DAG has a topological sort
    - If a directed graph has a topological sort, then it is a DAG

Topological sort

- In fact, let's look at (and prove) this interesting fact:
  - A directed graph is a DAG if and only if it has a topological sort.
  - Proof:

    We observe that there are two independent statements to prove:
    - A DAG has a topological sort
    - If a directed graph has a topological sort, then it is a DAG
  - (this is a normal aspect of if-and-only-if statements; proving them is really proving two statements)

Topological sort

- Let's start with the easy one:
  - If a graph has a topological sort, then it is a DAG.
  - Proof: (by contrapositive — that is, we prove the statement "if a graph is not a DAG, it can not have a topological sort")

Topological sort

- Let's start with the easy one:
  - If a graph has a topological sort, then it is a DAG.
  - Proof: (by contrapositive — that is, we prove the statement "if a graph is not a DAG, it can not have a topological sort")

    Assuming the graph is not a DAG means that we can find a cycle, say $\{v_1, v_2, \cdots, v_k, v_1\}$

    Since there is a path from v_1 to v_2, then v_1 must appear before v_2 in a topological sort.

    But there is also a path from v_2 to v_1, so v_2 must appear before v_1 in a topological sort.

Topological sort

We can not satisfy both conditions; therefore, the graph can not have a topological sort.

Topological sort

- For the other part, we prove by induction (on the number of vertices) that if a graph is a DAG, then it has a topological sort.

Topological sort

- For the other part, we prove by induction (on the number of vertices) that if a graph is a DAG, then it has a topological sort.
  - Base case: A graph with one vertex is a DAG, and it has a topological sort.

Topological sort

- For the other part, we prove by induction (on the number of vertices) that if a graph is a DAG, then it has a topological sort.
  - Base case: A graph with one vertex is a DAG, and it has a topological sort.
  - Induction hypothesis: A DAG with n vertices has a topological sort.

Topological sort

- For the other part, we prove by induction (on the number of vertices) that if a graph is a DAG, then it has a topological sort.
  - Base case: A graph with one vertex is a DAG, and it has a topological sort.
  - Induction hypothesis: A DAG with n vertices has a topological sort.
  - For the induction step, we must show that the induction hypothesis implies that a DAG with $n+1$ vertices must have a topological sort.

Topological sort

- Consider a graph with n+1 vertices.
  - Such a graph must have at least one vertex v_0 with in-degree 0 (can you prove this?).

Topological sort

- Consider a graph with n+1 vertices.
  - Such a graph must have at least one vertex v_0 with in-degree 0 (can you prove this?).
  - Remove that vertex to obtain a graph with n vertices.
    - Since the original graph had no cycles and we are removing edges (not adding), then the resulting graph must be a DAG.
    - By induction hypothesis, since it is a DAG with n vertices, then it has a topological sort.

Topological sort

- Consider a graph with *n*+1 vertices.
  - Thus, a topological sort can be constructed for the *n*+1 vertices DAG, by prepending v_0 to the topological sort of the n-vertices DAG.

    (we can definitely do that, since v_0 has in-degree 0, so no path exists from any other vertex to v_0, and this means that it can appear before any other vertex)

Topological sort

- A somewhat more “obvious” observation:
  - A topological sort is not necessarily unique.
    - There's one very solid argument to this in the previous proof ... anyone?

Topological sort

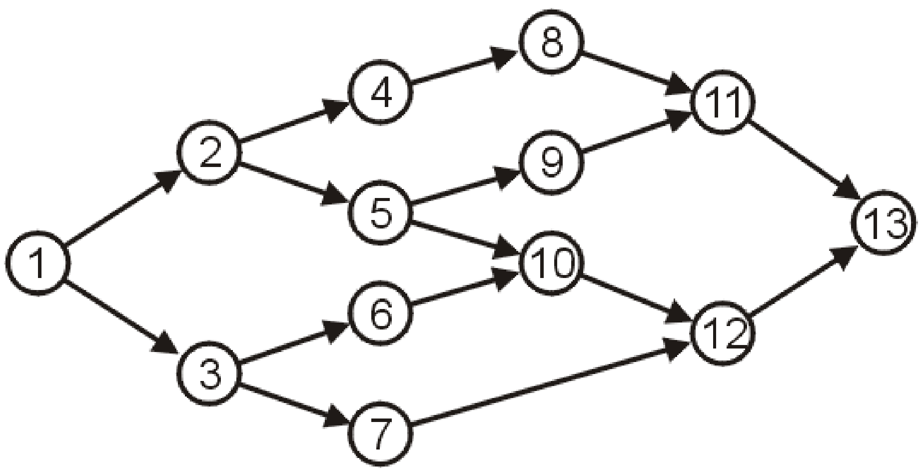
- A somewhat more “obvious” observation:
  - A topological sort is not necessarily unique.
    - There's one very solid argument to this in the previous proof ... anyone?
    - For one, there may be several vertices with in-degree 0, and either one of them can be the first one in a topological sort.

Topological sort

- Next, let's look at an algorithm to obtain a topological sort given a DAG.
  - The idea is that any vertex with in-degree 0 can be the first one in a topological sort.
  - We can look at it as follows: each time that we output one of those in-degree 0 vertices, we remove it from the graph.
    - That would in turn lead to creating additional vertices with in-degree 0, which we can now output.

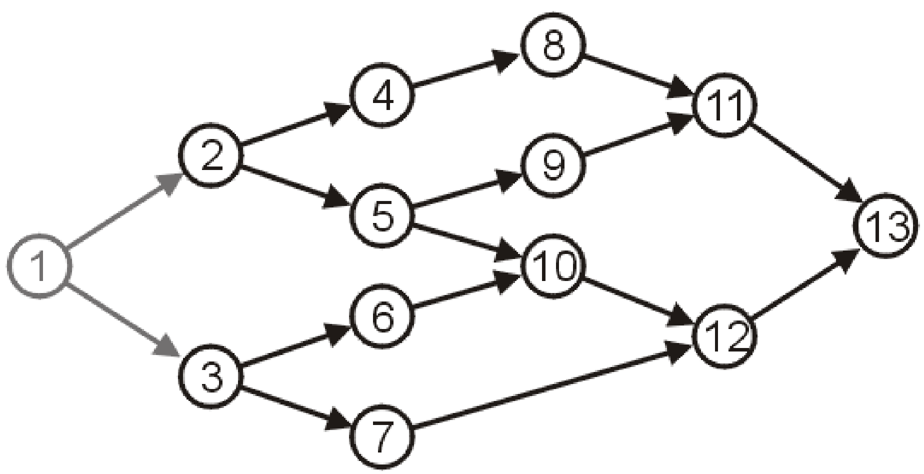
Topological sort

- An example:
  - There's only one vertex with in-degree 0 (vertex 1), so we start with that one (and think of it as removed from the graph):



Topological sort

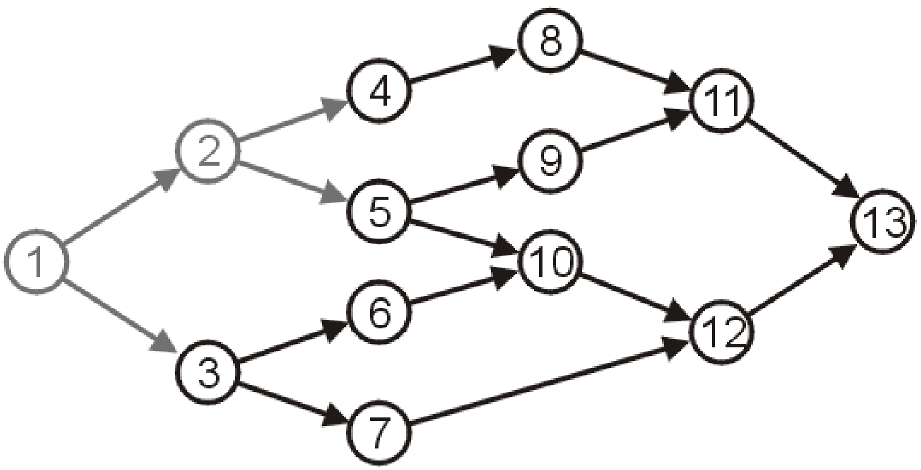
- An example:
  - As we "remove" 1, now 2 and 3 have in-degree 0, so the topological sort could continue with either one of these — we'll choose 2:



1

Topological sort

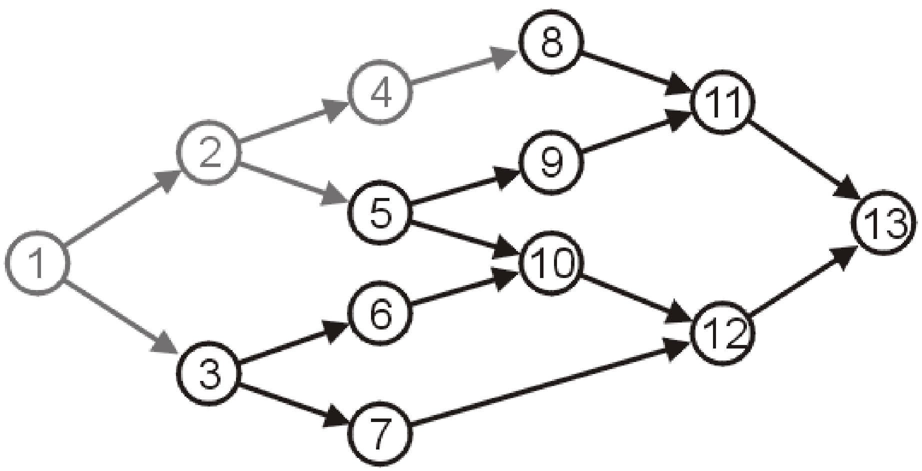
- An example:
  - Without 2, now 4 and 5 have in-degree 0, so the topological sort could continue with either 3, 4, or 5 — we'll choose 4:



1, 2

Topological sort

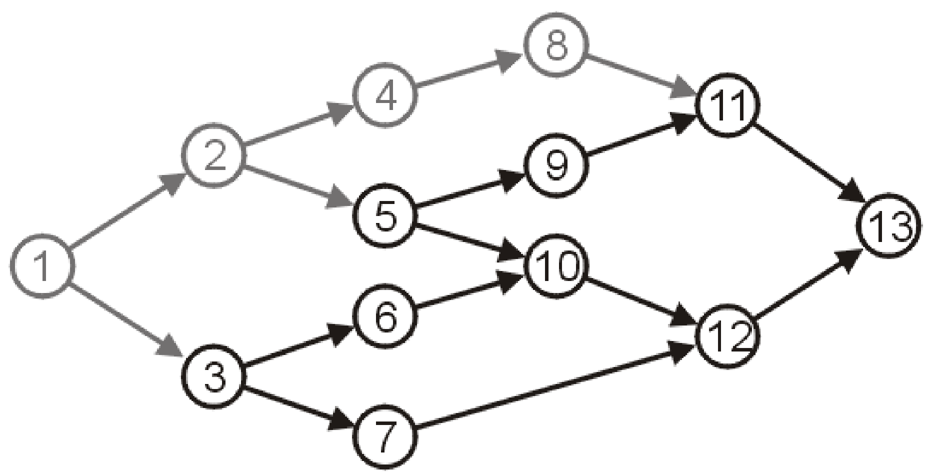
- An example:
  - We continue ...



1, 2, 4

Topological sort

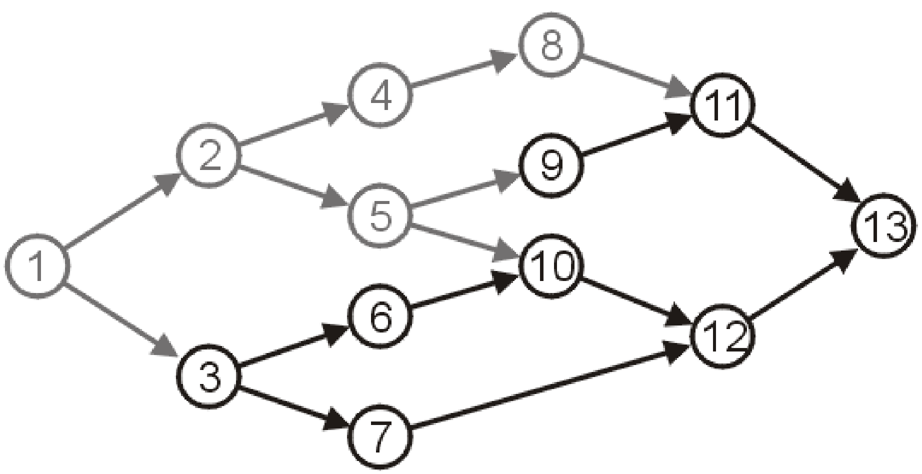
- An example:
  - We continue ...



1, 2, 4, 8

Topological sort

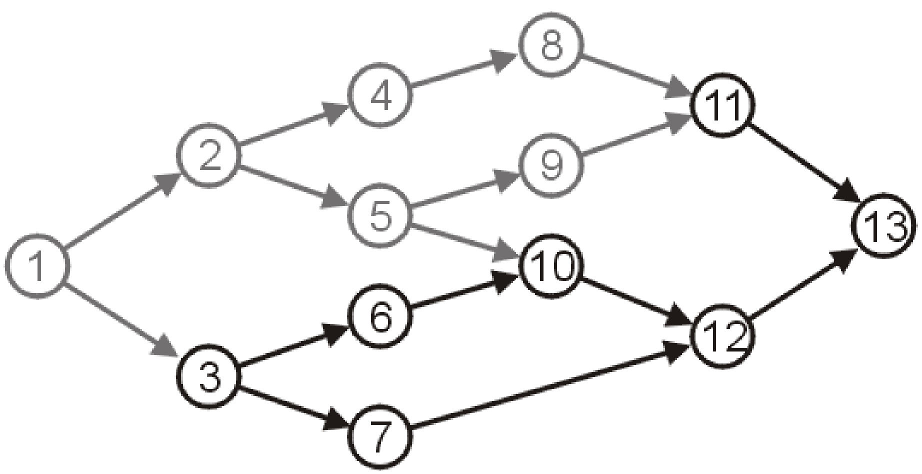
- An example:
  - We continue ...



1, 2, 4, 8, 5

Topological sort

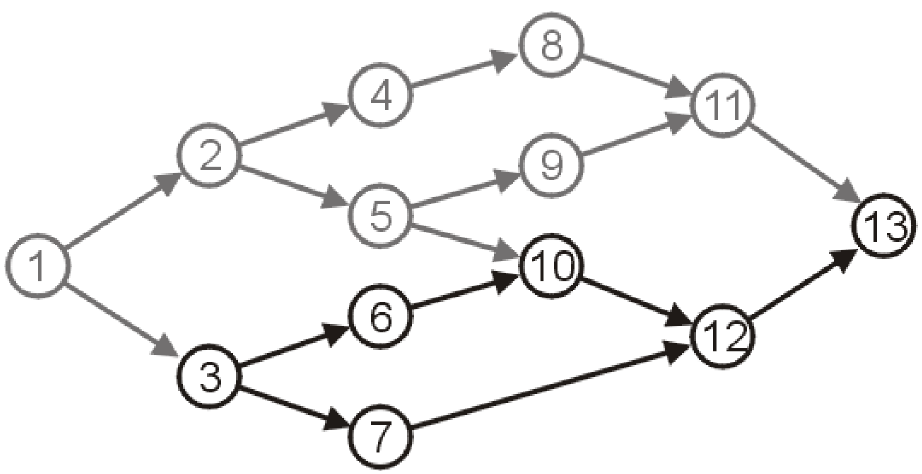
- An example:
  - We continue ...



1, 2, 4, 8, 5, 9

Topological sort

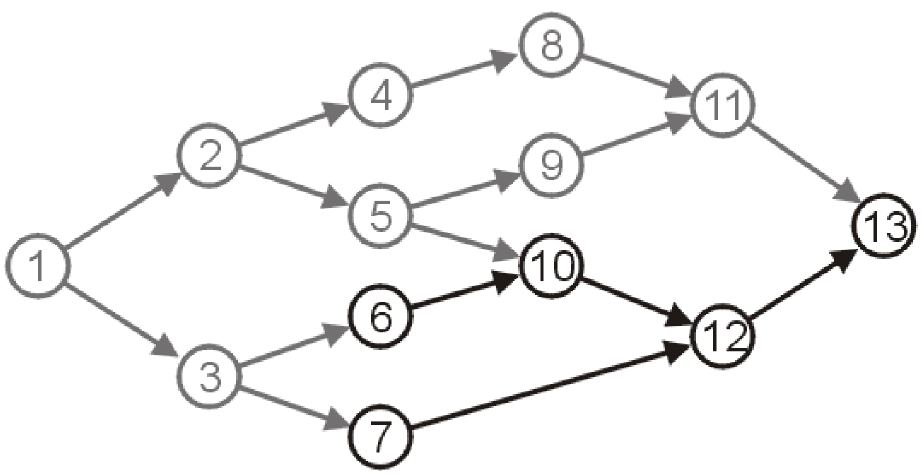
- An example:
  - We continue ...



1, 2, 4, 8, 5, 9, 11

Topological sort

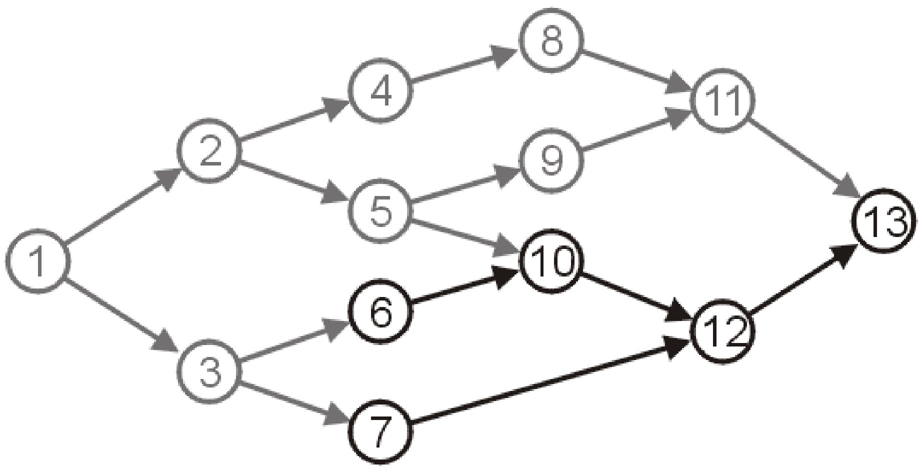
- An example:
  - We continue ...



1, 2, 4, 8, 5, 9, 11, 3

Topological sort

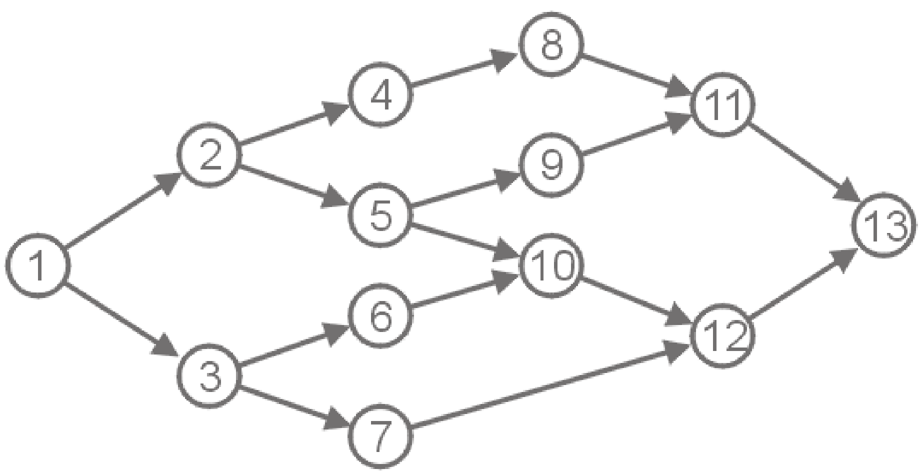
- An example:
  - We continue ...



1, 2, 4, 8, 5, 9, 11, 3 ⋯ etc.

Topological sort

- An example:
  - At this point, the output is complete.



1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13

Topological sort

- The fact that a topological sort is not unique should have become quite clear from this example — anyone?

Topological sort

- The fact that a topological sort is not unique should have become quite clear from this example — anyone?
  - At several points in the process, we had to choose one among several equally valid candidates; different choices would have produced different topological sorts.

Topological sort

- A typical implementation uses an array of in-degrees, and optionally a queue (for efficiency)

Topological sort

- A typical implementation uses an array of in-degrees, and optionally a queue (for efficiency)
  - We start by initializing the table of in-degrees (how do we do this efficiently? A single pass through the list of vertices, perhaps?)

Topological sort

- A typical implementation uses an array of in-degrees, and optionally a queue (for efficiency)
  - We start by initializing the table of in-degrees (how do we do this efficiently? A single pass through the list of vertices, perhaps?)
    - And BTW, why the big deal with doing this efficiently?? How would we do it inefficiently?

Topological sort

- Obtaining the in-degrees of every vertex inefficiently is quite easy — for each vertex, determine its in-degree by visiting every other vertex to count how many this vertex is adjacent to (quadratic run time).

Topological sort

- A more reasonable approach is: initialize all in-degrees in the array to 0. Visit each vertex, and increase by 1 the in-degree of each of the vertices that are adjacent to the one being visited.
  - What's the run time of this?

Topological sort

- A more reasonable approach is: initialize all in-degrees in the array to 0. Visit each vertex, and increase by 1 the in-degree of each of the vertices that are adjacent to the one being visited.
  - What's the run time of this?
  - Would you agree if I said it is $\Theta(|V| + |E|)$?

Topological sort

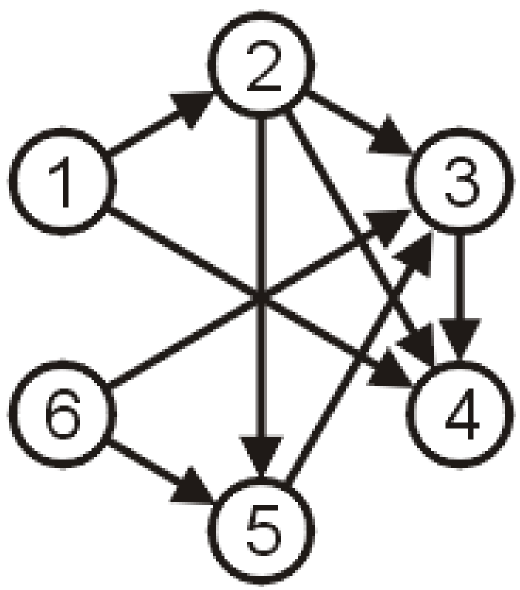
- A more reasonable approach is: initialize all in-degrees in the array to 0. Visit each vertex, and increase by 1 the in-degree of each of the vertices that are adjacent to the one being visited.
  - What's the run time of this?
  - Would you agree if I said it is $\Theta(|V|+|E|)$?
  - In fact, we'll see that this is the run time for the topological sort (i.e., for the whole procedure)

Topological sort

- Let's look at an example, directly from Prof. Harder's slides.

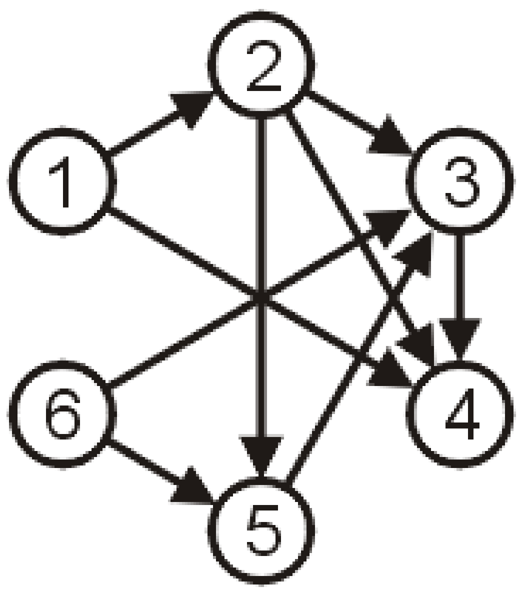
Example

Consider the following DAG with six vertices



Example

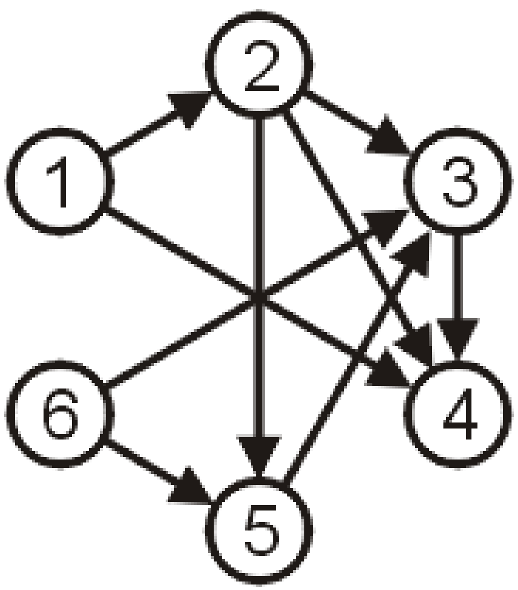
Let us define the table of in-degrees
(or more likely, copy it)



Vertex	In-degree
1	0
2	1
3	3
4	3
5	2
6	0

Example

And a queue into which we can insert vertices 1 and 6



Vertex	In-degree
1	**0**
2	1
3	3
4	3
5	2
6	**0**

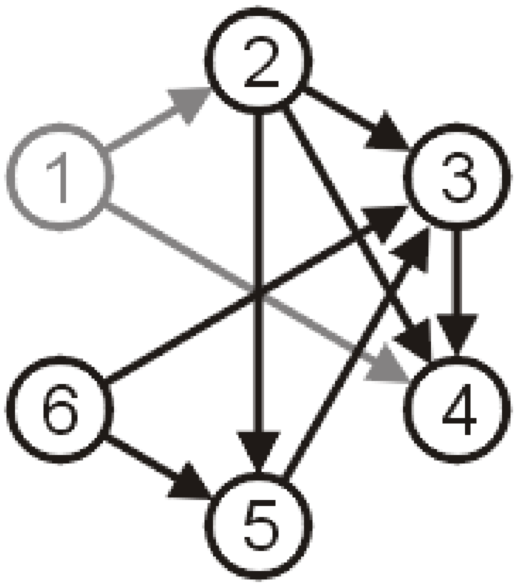
Queue

1 **6**

Example

We dequeue the head (1), decrement the in-degree of all adjacent vertices: 2 and 4

- 2 is decremented to zero: enqueue 2



Vertex	In-degree
1	0
2	**0**
3	3
4	**2**
5	2
6	0

Queue

6 2

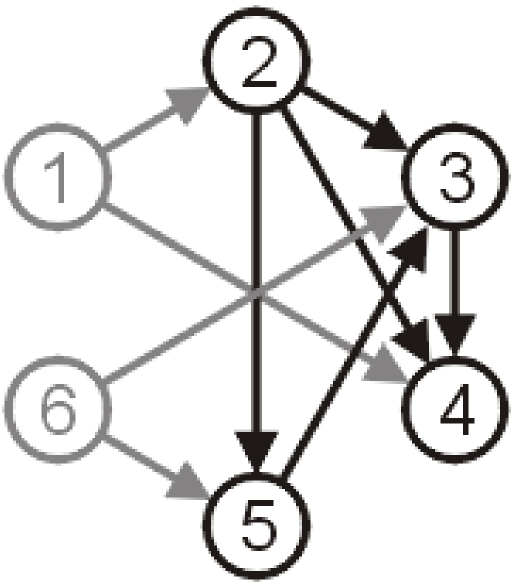
Sort

1

Example

We dequeue 6 and decrement the in-degree of all adjacent vertices

- Neither is decremented to zero



Vertex	In-degree
1	0
2	0
3	**2**
4	2
5	**1**
6	0

Queue

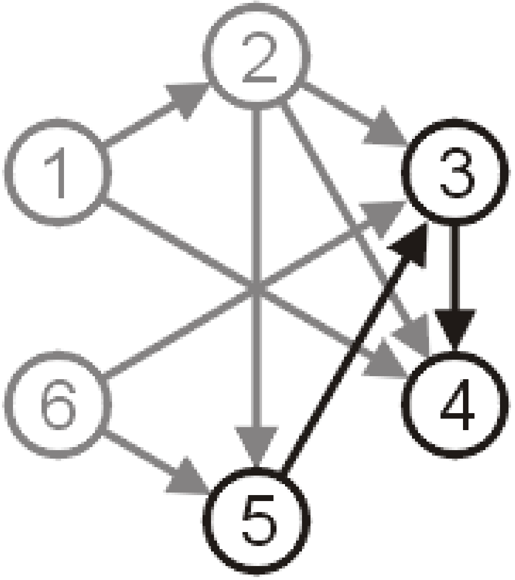
2

Sort

1, 6

Example

We dequeue 2, decrement, and enqueue vertex 5



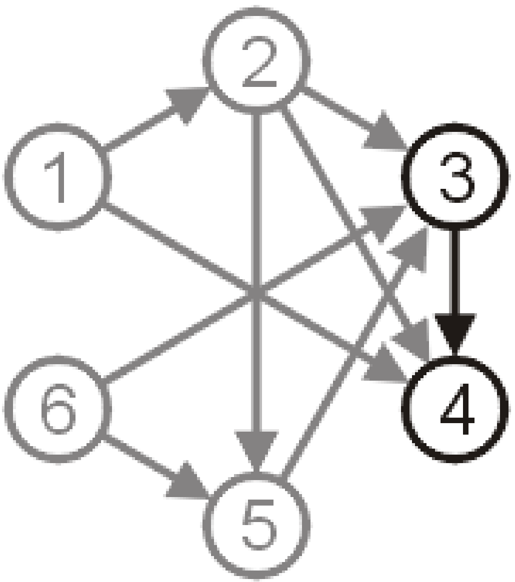
Vertex	In-degree
1	0
2	0
3	**1**
4	**1**
5	**0**
6	0

Queue
5

Sort
1, 6, 2

Example

We dequeue 5, decrement, and enqueue vertex 3



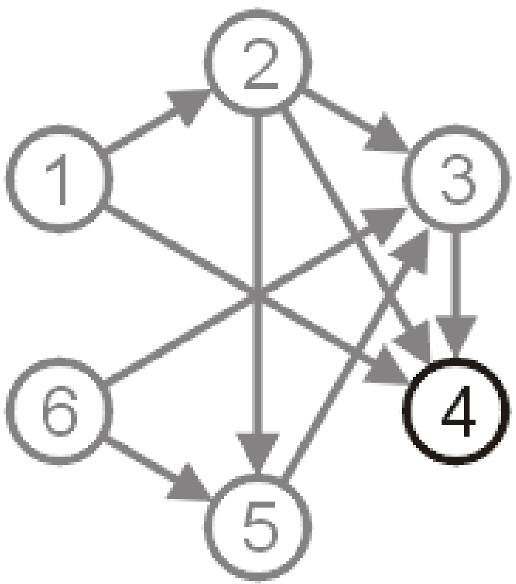
Vertex	In-degree
1	0
2	0
3	**0**
4	1
5	0
6	0

Queue
3

Sort
1, 6, 2, 5

Example

We dequeue 3, decrement 4, and add 4 to the queue



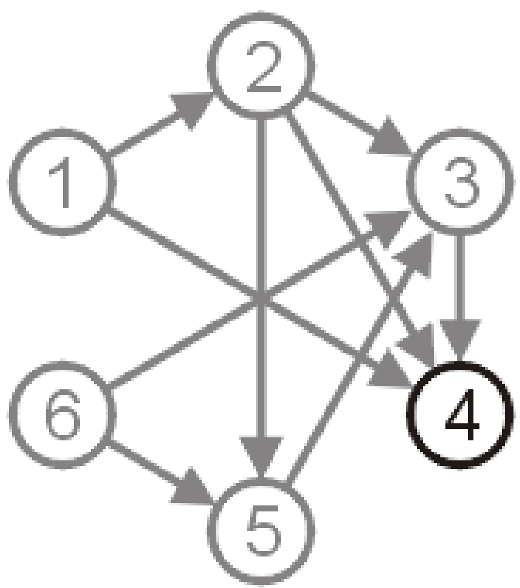
Vertex	In-degree
1	0
2	0
3	0
4	**0**
5	0
6	0

Queue
4

Sort
1, 6, 2, 5, 3

Example

We dequeue 4, there are no adjacent vertices to decrement the in degree



Vertex	**In-degree**
1	0
2	0
3	0
4	0
5	0
6	0

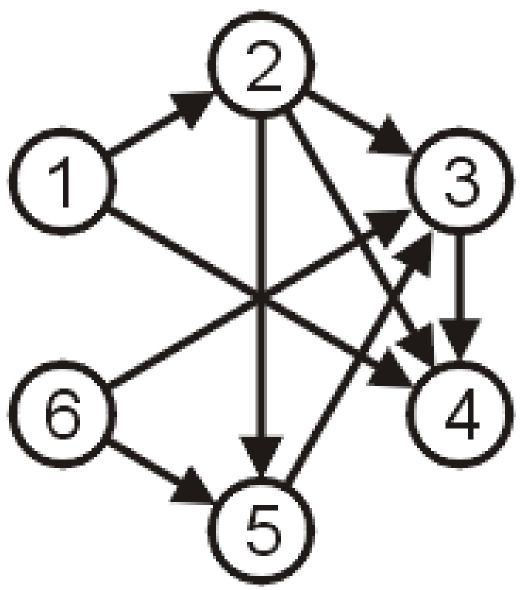
Queue

Sort
1, 6, 2, 5, 3, 4

Example

The queue is now empty, so a topological sort is

1, 6, 2, 5, 3, 4



Topological sort

- And BTW ... How do we implement the graph itself?

Topological sort

- And BTW ... How do we implement the graph itself?
  - We typically don't go for a tree-like implementation of a node:
    - Too much overhead: Potentially *very* large number of associations, but actual graphs tend to contain a small fraction of that maximum.

Topological sort

- And BTW ... How do we implement the graph itself?
  - Two typical approaches are:
    - Adjacency lists
    - Adjacency matrix

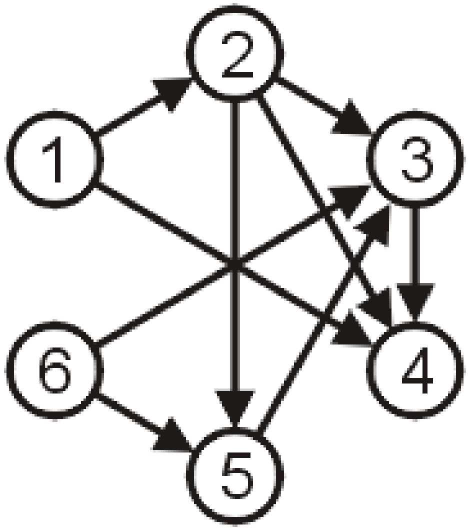
Topological sort

- With *adjacency lists*, vertices are associated with a number between 0 and $|V|-1$
- An array of adjacencies is defined — each element of the array is a list (either a dynamic array or a linked list) of the vertices adjacent to the vertex corresponding to that subscript.

Topological sort

- Ajacency list — example:

$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(1,2), (1,4), (2,3), (2,4), (2,5), (3,4), (5,3), (6,3), (6,5)\}$$

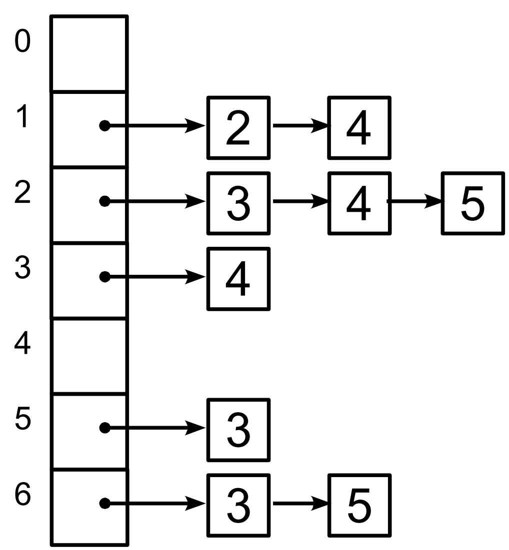


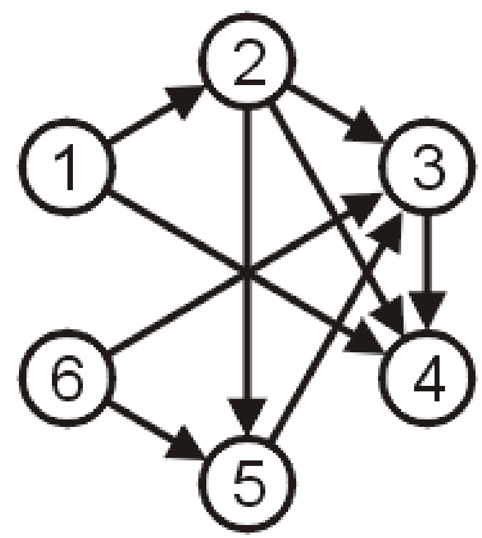
Topological sort

- Ajacency list — example:

$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(1,2), (1,4), (2,3), (2,4), (2,5), (3,4), (5,3), (6,3), (6,5)\}$$

Representation:





Topological sort

- With an adjacency matrix, as the name suggests, we define a $|V| \times |V|$ array of booleans (if unweighted graph) or doubles (if weighted graph) where element at [row][col] indicates whether vertex col is adjacent to vertex row.

Topological sort

- Adjacency list vs. Adjacency matrix — which approach is better?

Topological sort

- Adjacency list vs. Adjacency matrix — which approach is better?
  - Not surprisingly, it depends on the situation:
    - If there are very few edges, then the adjacency matrix is inefficient in memory usage.
    - However, if there are many edges, then the extra complexity in the adjacency list is not justified.

Topological sort

- Adjacency list vs. Adjacency matrix — which approach is better?
  - Not surprisingly, it depends on the situation:
    - If there are very few edges, then the adjacency matrix is inefficient in memory usage.
    - However, if there are many edges, then the extra complexity in the adjacency list is not justified.
    - With an adjacency list, we can efficiently iterate over all the adjacent vertices (why not with an adjacency matrix?)
    - However, with an adjacency matrix we can determine in $\Theta(1)$ whether two given vertices are adjacent.

Summary

- During today's class, we:
  - Looked at Topological sort for Directed Acyclic Graphs (DAGs)
  - Presented an algorithm to implement this operation.
  - Saw some of its applications.
  - Briefly looked into implementation strategies:
    - Adjacency list
    - Adjacency matrix