香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures Lecture 19: DAG checking, topological sort

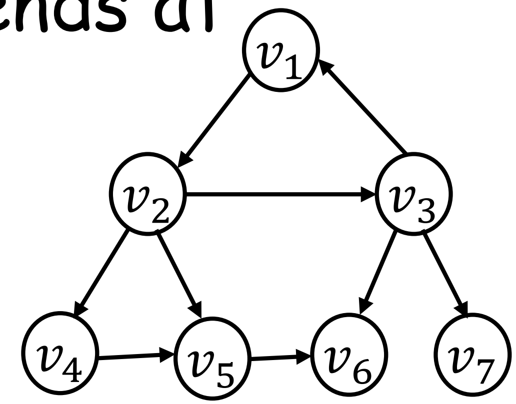
Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Directed acyclic graph (DAG)
  - DAG checking
  - Topological sort

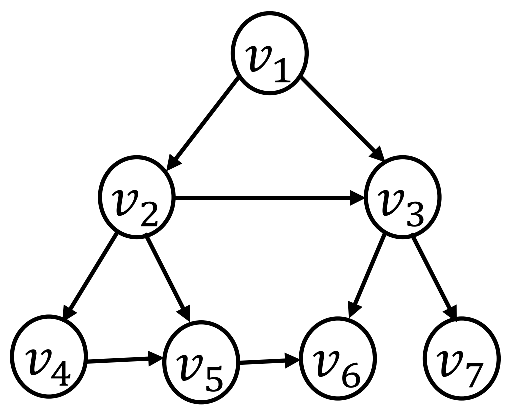
Directed Acyclic Graph (DAG)

- Cycle: A simple path that starts and ends at the same node
  - In directed graph G
    - Path $P = (v_1, v_2, v_3, v_1)$ is a cycle



G

- Directed acyclic graph (DAG)
  - A directed graph that contains no cycles



A DAG

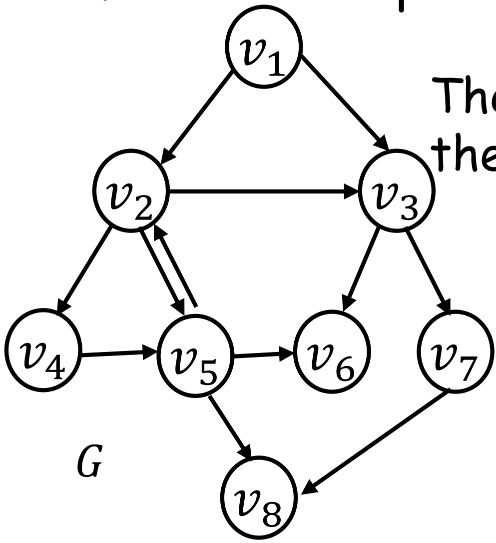
DAG Checking: Using DFS

- Doing Depth-first search on the entire graph G
  - The DFS we learned in last lecture has an input source s

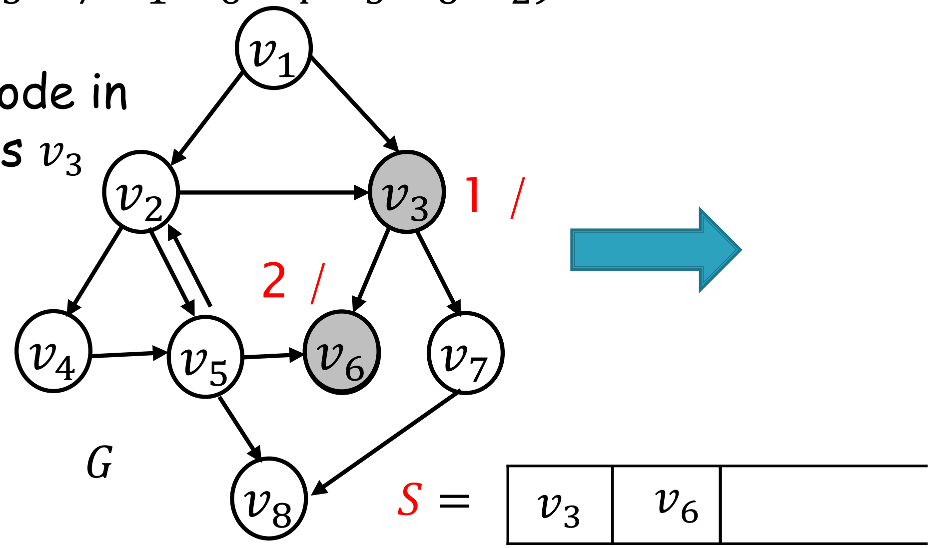
- To apply to the entire graph:
  - Randomly generate a permutation of the nodes and repeat the following until there is no white node
    - Pick the first white node s in the permutation and do DFS (during DFS, we will color nodes, and record timestamps)

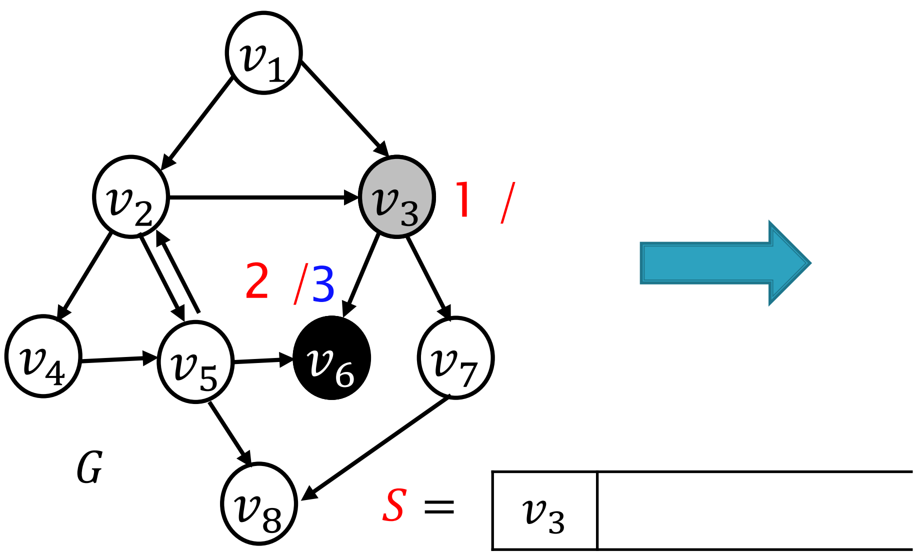
A Running Example

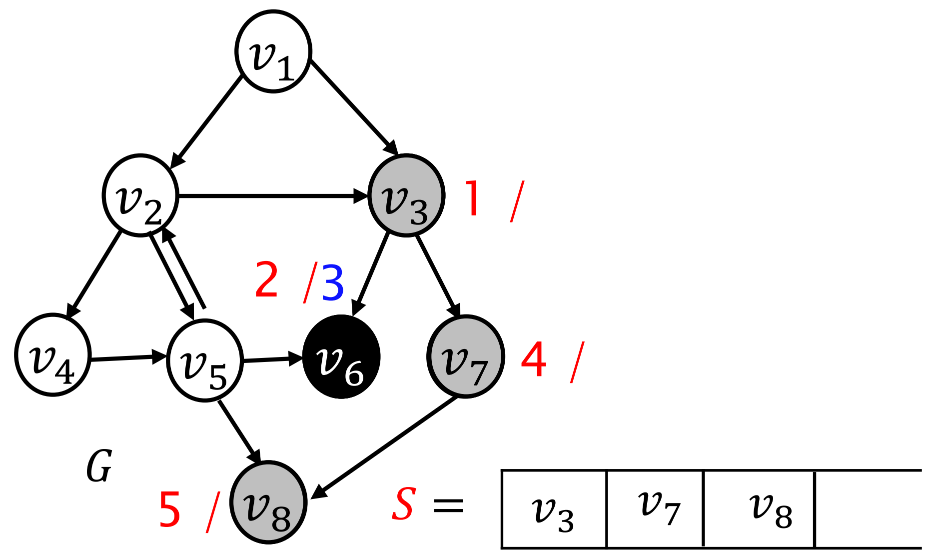
Assume that the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2)$



The first white node in the permutation is v_3



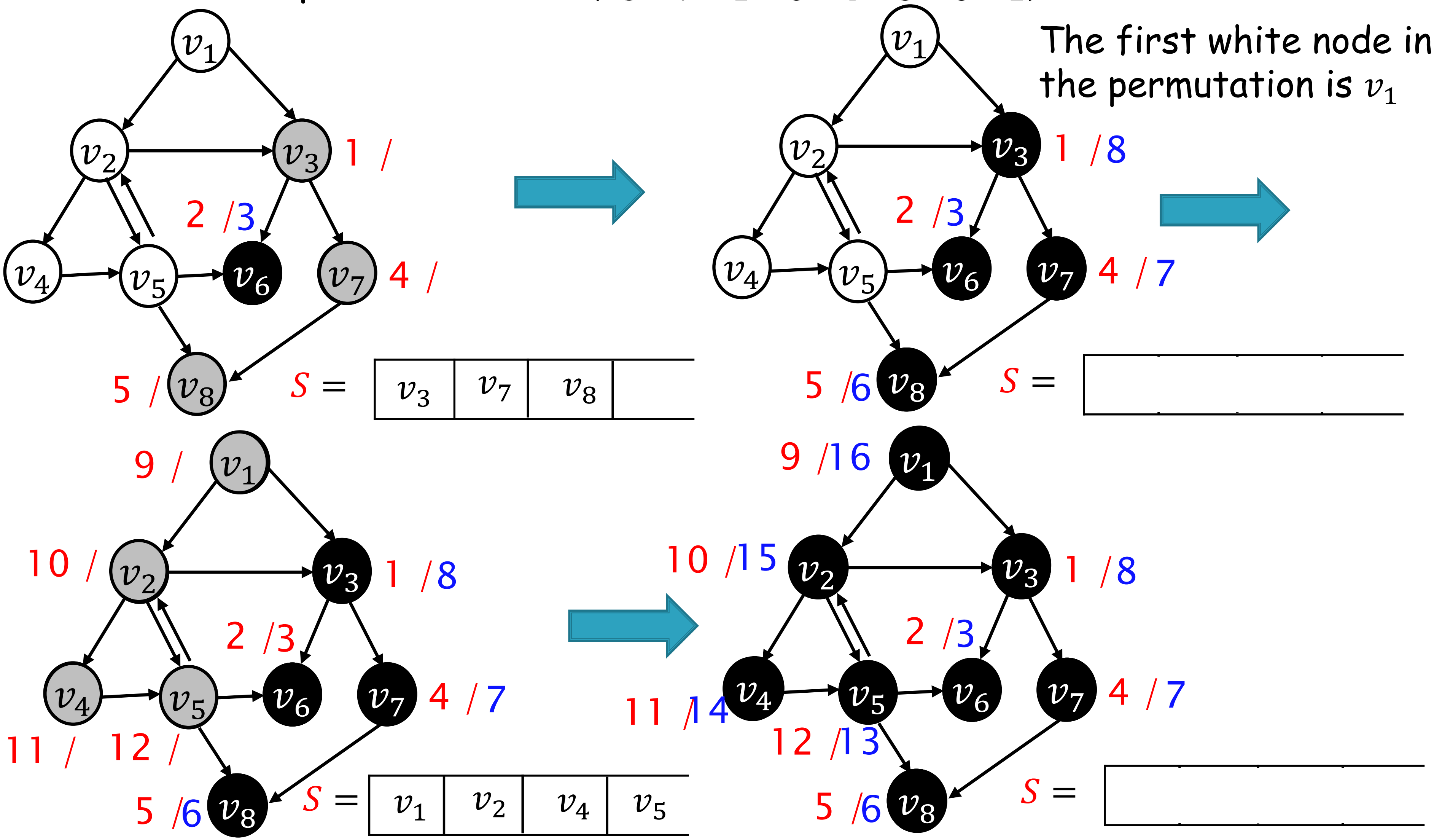




A Running Example

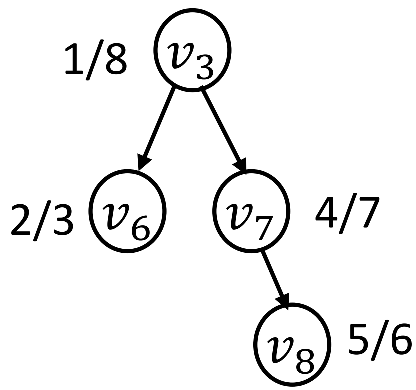
Assume the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2)$

The first white node in the permutation is v_1

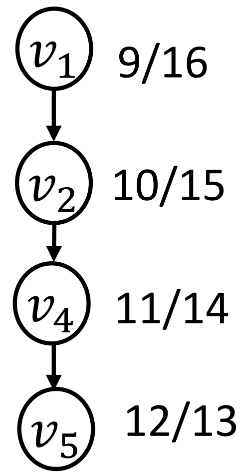


Edge Classifications (i)

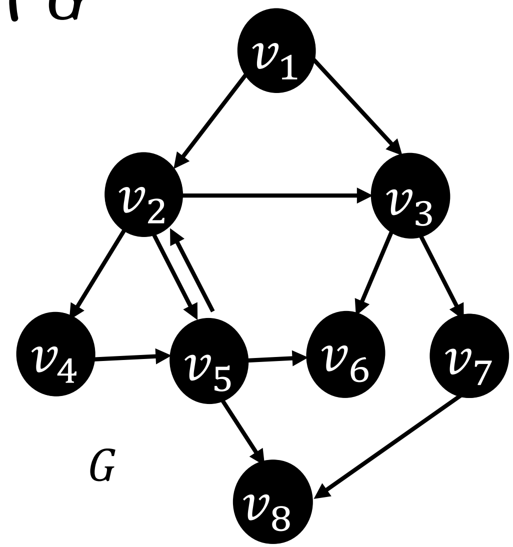
- Results of the DFS-trees on graph G



DFS-Tree
rooted at v_3



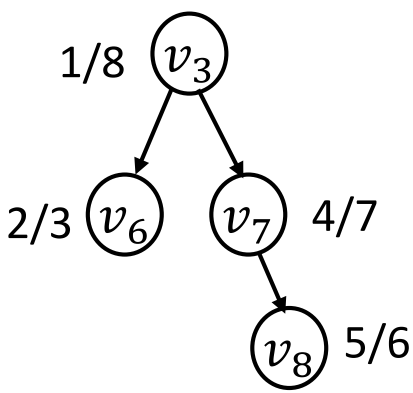
DFS-Tree
rooted at v_1



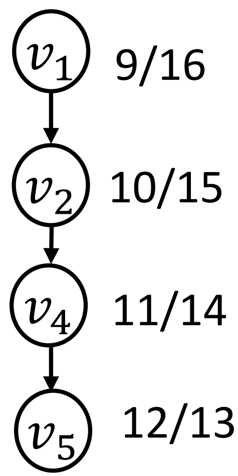
  - v_3 is an ancestor of v_8 in the DFS tree rooted at v_3
  - v_5 is an descendant of v_1 in the DFS tree rooted at v_1
  - Neither v_1 or v_3 is the descendant of the other

Edge Classifications (ii)

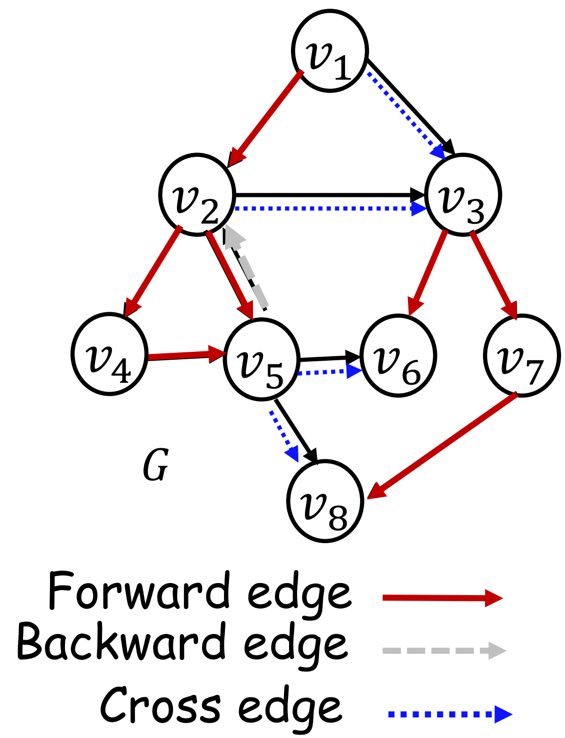
- Assume we have done DFS on graph G. Let $\langle u, v \rangle$ be an edge in G. It can be classified into three types:
  - Forward edge: if u is an ancestor of v in one of the DFS-trees
  - Backward edge: if u is an descendant of v in one of the DFS-trees
  - Cross edge: if none of the above happens



DFS-Tree rooted at v_3

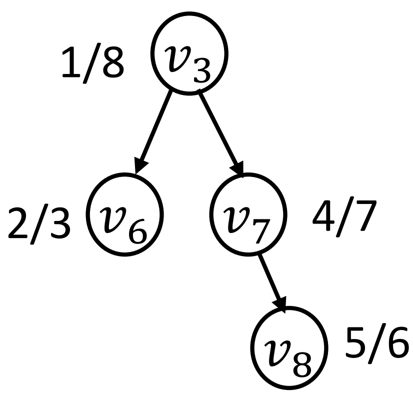


DFS-Tree rooted at v_1

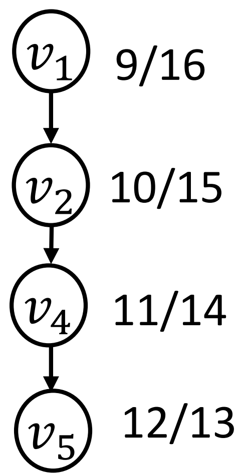


Recap: Interval Property

- Interval $I(u)$ of node u is $[u.d, u.f]$, where $u.d$ is the first discovery time and $u.f$ is the finish time
  - We will only have three cases for two nodes u and v
    - $I(u) \subset I(v)$, u is the descendant of v
    - $I(v) \subset I(u)$, v is the descendant of u
    - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other.



DFS-Tree
rooted at v_3



DFS-Tree
rooted at v_1

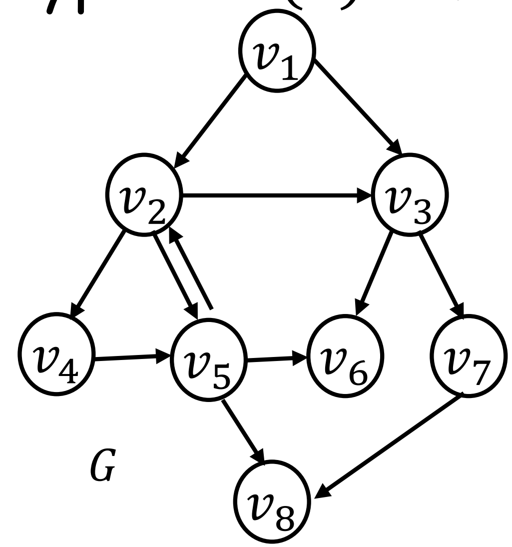
$I(v_5) \subset I(v_1)$: v_5 is the descendant of v_1

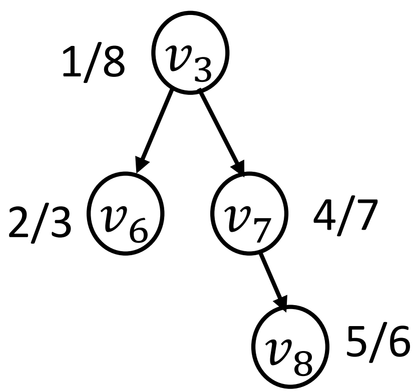
$I(v_6) \cap I(v_7) = \emptyset$:neither one is the descendant of the other

How about v_3 and v_8?
How about v_3 and v_1?

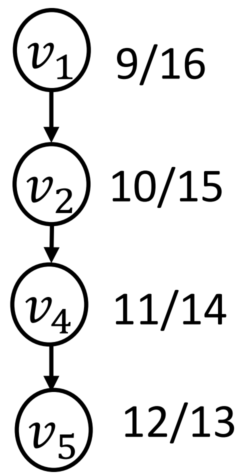
Cost for Edge Classifications

- For an edge $\langle u, v \rangle$, we can check edge type in $O(1)$ time given the interval information
  - $I(u) \subset I(v)$, backward edge
  - $I(v) \subset I(u)$, forward edge
  - $I(u) \cap I(v) = \emptyset$, cross edge





DFS-Tree rooted at v_3



DFS-Tree rooted at v_1

$\langle v_2, v_3 \rangle$: $I(v_2) = [10, 15]$,
$I(v_3) = [1,8]$
$I(v_2) \cap I(v_3) = \emptyset$. Cross edge

How about $\langle v_2, v_5 \rangle$ and $\langle v_5, v_2 \rangle$?

Cycle Theorem

Theorem 1: Given the DFS result on graph G, then G contains a cycle if and only if there is a backward edge in the DFS result on G.

Proof: (i) there is a backward edge $\langle u, v\rangle$, then G contains a cycle. This part can be proved according to the definition and will be left as exercise.
(ii) Prove that if there is a cycle, then there will exist a backward edge. Assume that the cycle is $(v_1, v_2, v_3, \cdots v_l, v_1)$. Then actually, we know path $(v_2, v_3, \cdots, v_l, v_1, v_2)$ is also a cycle, and so on for the other paths starting from $v_3, v_4, \cdots v_l$.

Assume that v_i is the first node to be pushed onto the stack when doing DFS from a source s. Then, since there is a path from v_i to any other nodes $v_1, v_2, \cdots, v_{i-1}, v_{i+1}, \cdots v_l$, all these nodes will be visited during this DFS traversal with source s, and will be descendant of v_i. Therefore, we have an edge $\langle v_{i-1}, v_i\rangle$, and v_{i-1} is an descendant of v_i, which is a backward edge according to the definition. Proof done.

Cycle Detection: Putting it All Together

- Step 1: Do DFS traversal on graph G
  - Time complexity: $O(n+m)$ (permutation can be done in $O(n)$)

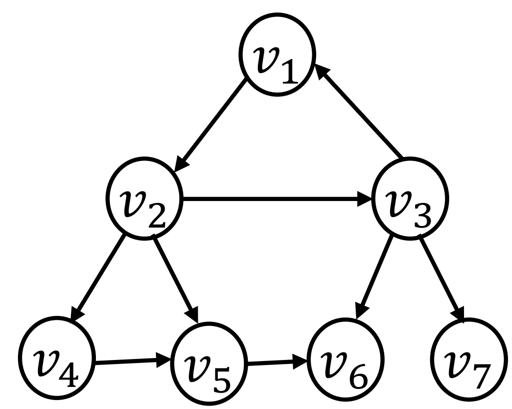
- Step 2: Classify edges according to the interval of each node derived with DFS
  - Time complexity: $O(m)$

- Step 3: If there exists a backward edge, G contains a cycle, otherwise, G is a directed acyclic graph

- Total time complexity:
  - $O(n+m)$

Practice

- Given the input graph G, and assume that the permutation generated for the nodes is: $(v_3, v_2, v_4, v_5, v_7, v_6, v_1)$
  - Verify if the graph is a DAG by using DFS step by step
  - In your solution, you should explicitly output the type of each edge

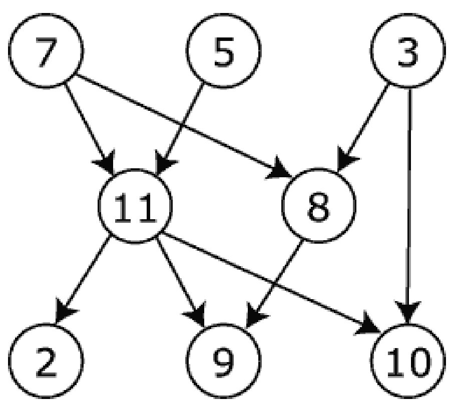


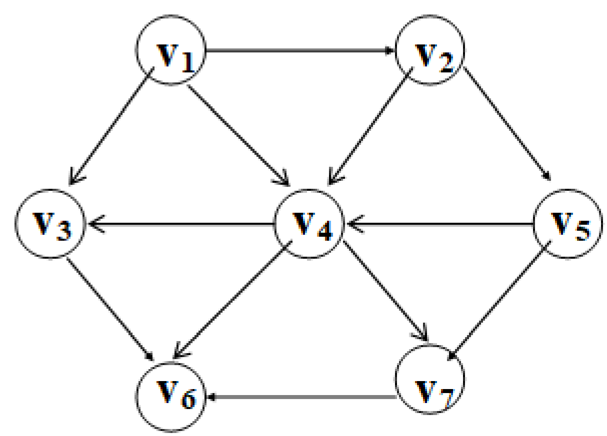
G

Topological sort

Topological Sort

- An ordering of all vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering
- If there is no path between v_i and v_j, then any order between them is fine
- Applications: job scheduling, logistics planning, course selection for each term





Topological Sort

- Topological ordering is not possible if there is a cycle in the graph
- A DAG has at least one topological ordering
- A simple algorithm
  - Compute the indegree of all vertices from the adjacency information of the graph
  - Find any vertex with no incoming edges
  - Print this vertex, and remove it, and its edges
  - Apply this strategy to the rest of the graph

Topological Sort

```
/* Assume that the graph is already read into an adjacency list and that the
indegrees are computed and placed in an array */
void topsort () {
        for (int counter=0; counter<numVertex; counter++) {
                Vertex v = FindNewVertexOfInDegreeZero (); //check all vertices
                if (v == null) {
                        Error("Cycle Found"); return;
                }
                v.topNum = counter;
                for each Vertex w adjacent to v
                        w.indegree--;
        }
}
```

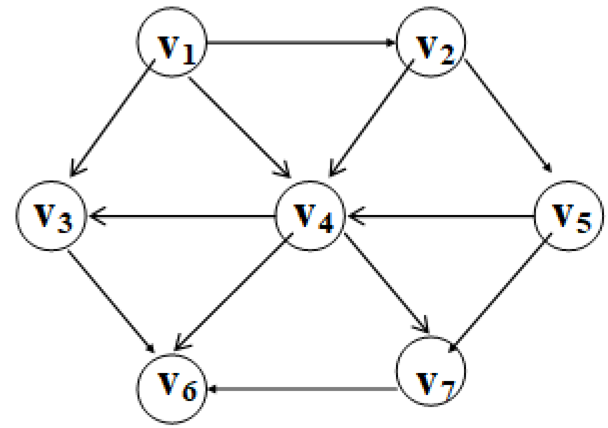
Running time is $O(|V|^2)$

Topological sort

- An improved algorithm
  - Keep all the unassigned vertices of indegree 0 in a queue
  - While queue is not empty
  - Remove a vertex in the queue
  - Decrease the indegrees of all adjacent vertices
  - If the indegree of an adjacent vertex becomes 0, enqueue the vertex
  - Running time is $O(|E|+|V|)$

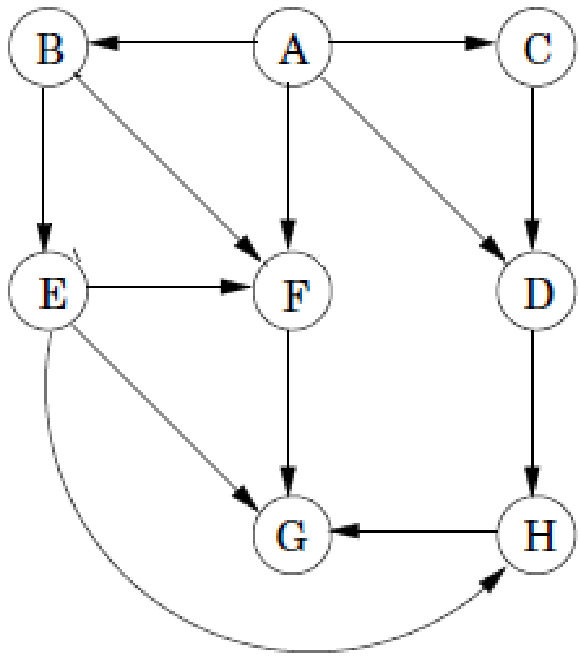
Topological sort

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6



Exercise

- Compute the topological sort for the following graph



Recommended Reading

- Reading this week
  - Textbook Chapters 22.3-22.4, 24.3
- Next week
  - Final exam: 16:00-18:00pm, July 26