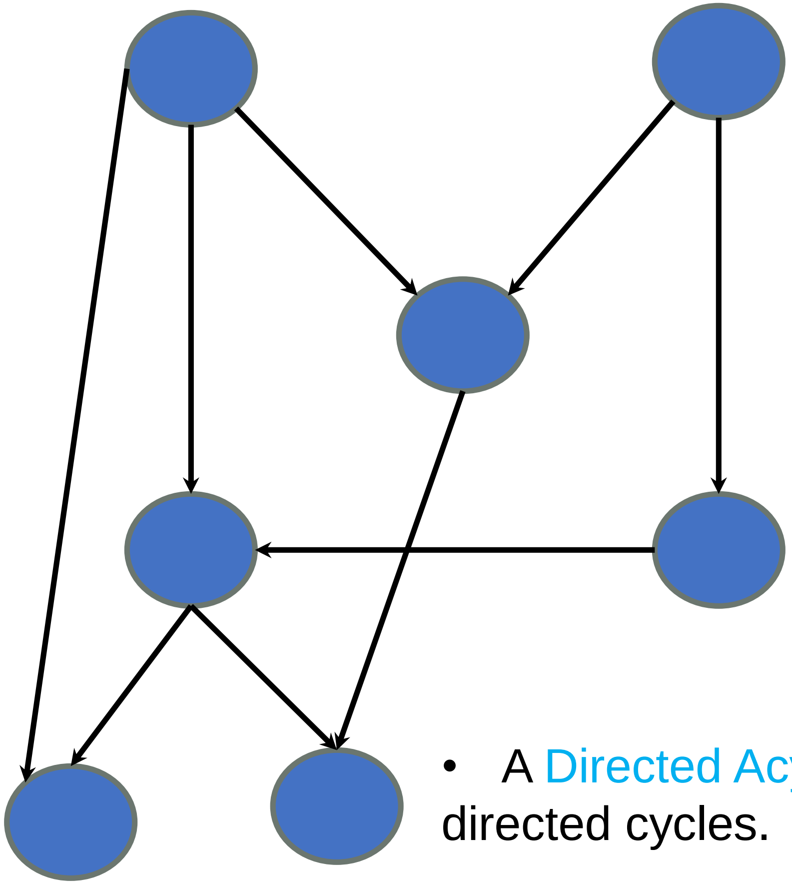
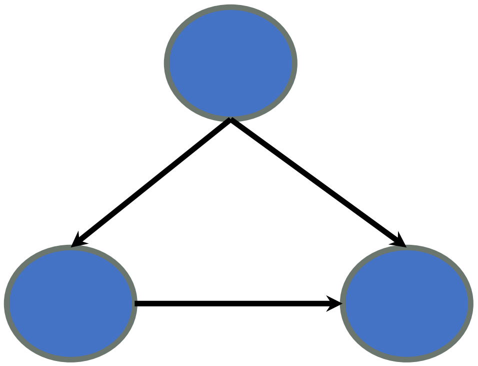
CSE 215: Data Structures and Algorithms II

Topological Sorting

Strongly Connected Components

Directed Acyclic Graph





- A Directed Acyclic Graph or DAG is a directed graph with no directed cycles.

Topological Sort

- A *topological sort* of a DAG is

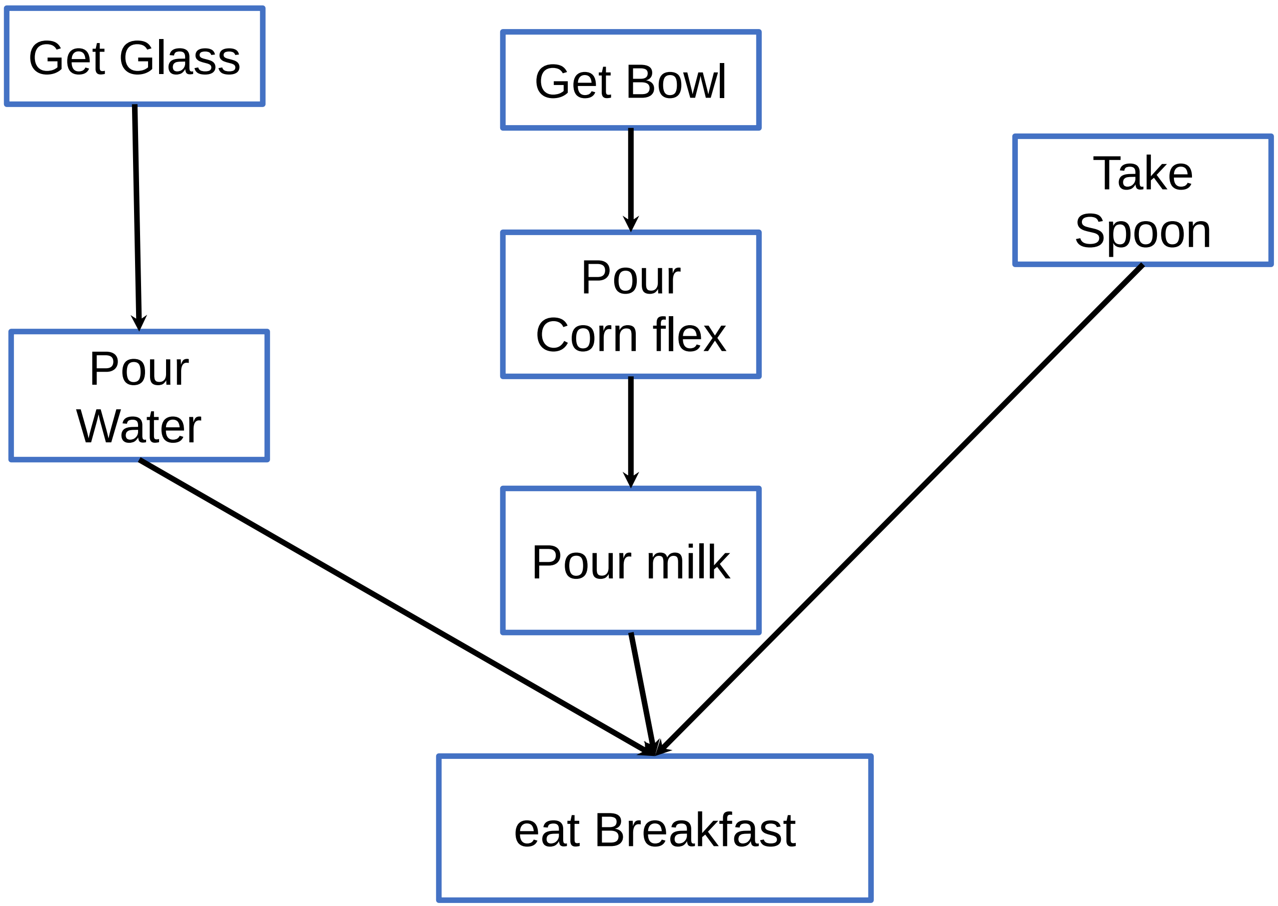
  a linear ordering of all vertices of the graph *G* such that vertex *u* comes before vertex *v* if (*u*, *v*) is an edge in *G*.
- DAG indicates precedence among events:

  events are graph vertices, edge from *u* to *v* means event *u* has precedence over event *v*
- Real-world example:
  - getting dressed
  - course registration
  - tasks for eating meal

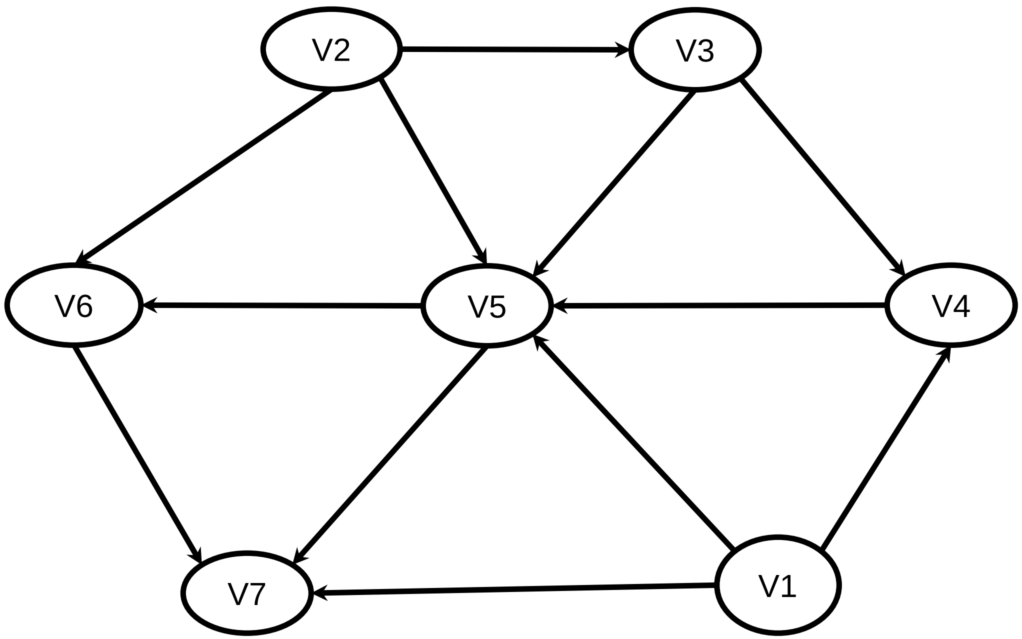
Precedence Example

- Tasks that have to be done to eat breakfast:
  - get glass, pour juice, get bowl, pour cereal, pour milk, get spoon, eat.
- Certain events must happen in a certain order (ex: get bowl before pouring milk)
- For other events, it doesn't matter (ex: get bowl and get spoon)

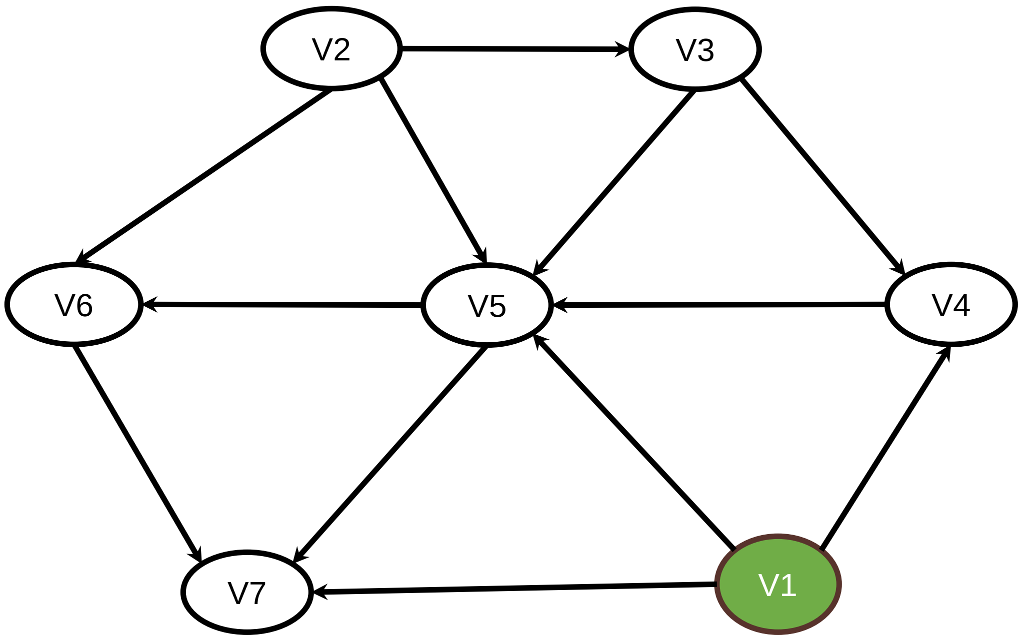
Precedence Example



Topological Sort: Using in-degree

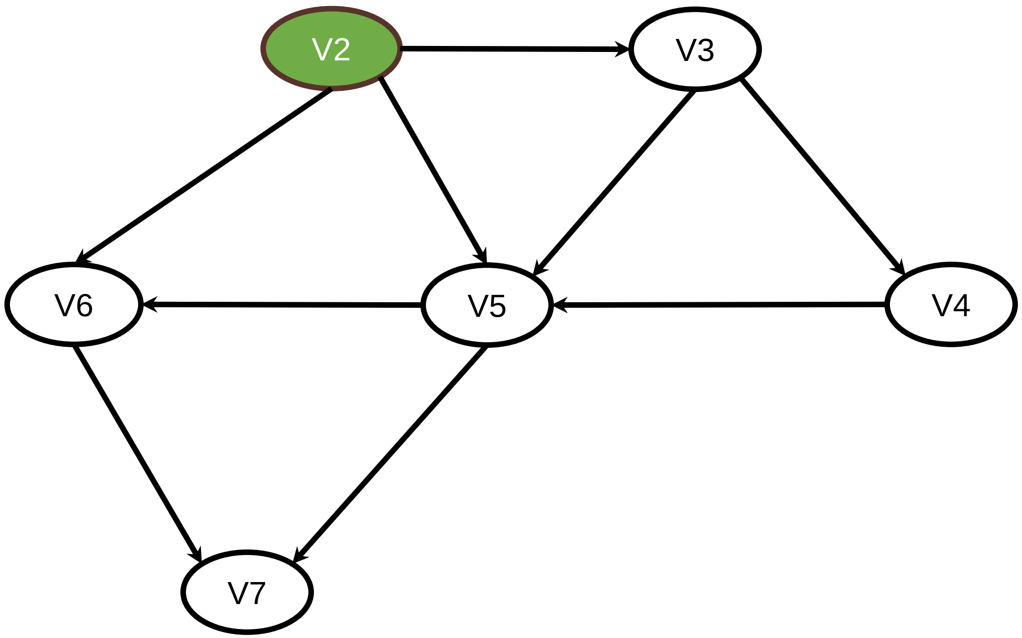


Topological Sort: Using in-degree



Delete the vertex whose in-degree 0. (v1 or v2)

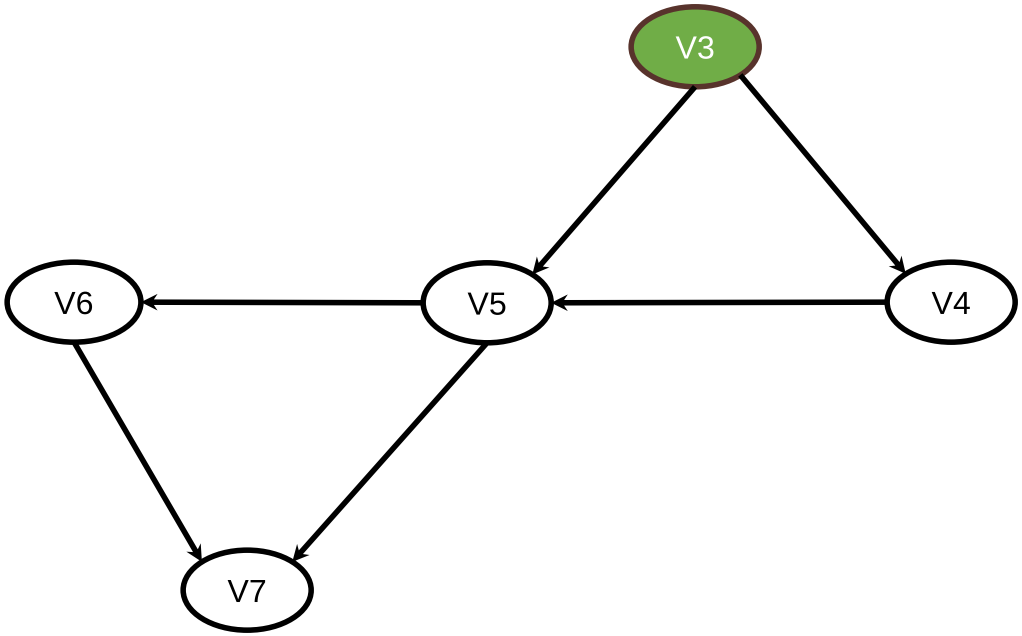
Topological Sort: Using in-degree



T: v1

Delete the vertex whose in-degree 0. (v2)

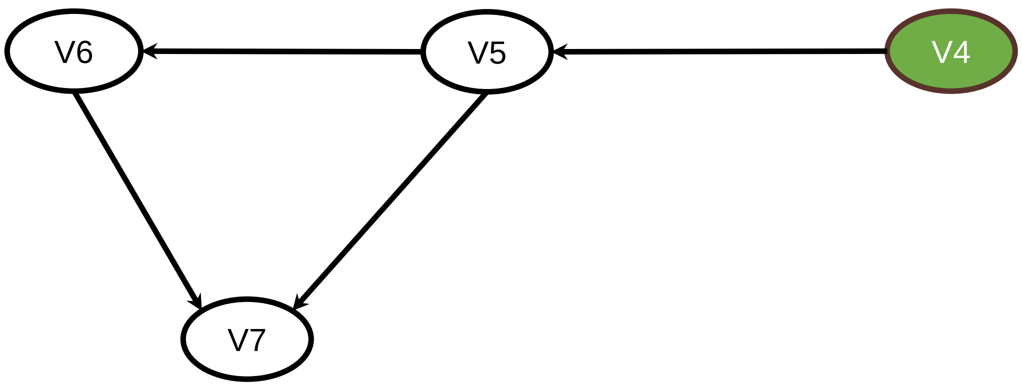
Topological Sort: Using in-degree



T: V1 V2

Delete the vertex whose in-degree 0. (v3)

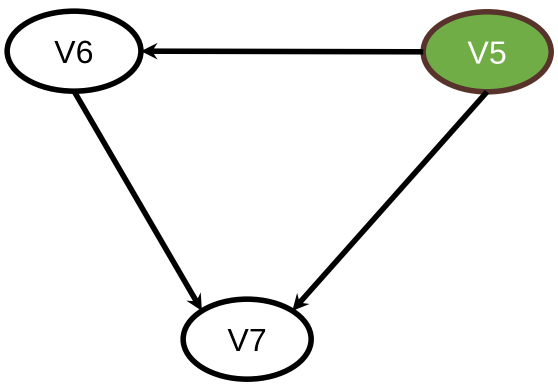
Topological Sort: Using in-degree



T: v1 v2 v3

Delete the vertex whose in-degree 0. (v4)

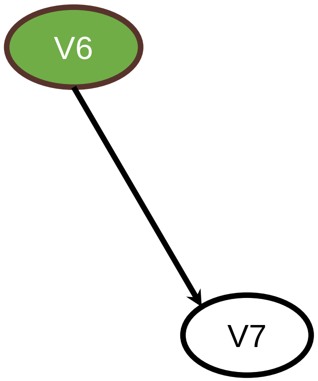
Topological Sort: Using in-degree



T: v1 v2 v3 v4

Delete the vertex whose in-degree 0. (v5)

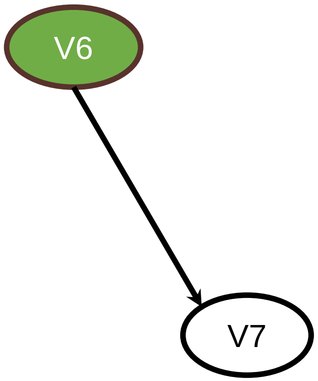
Topological Sort: Using in-degree



T: v1 v2 v3 v4 v5

Delete the vertex whose in-degree 0. (v6)

Topological Sort: Using in-degree



T: v1 v2 v3 v4 v5 v6

Delete the vertex whose in-degree 0. (v6)

Topological Sort: Using in-degree

T: v1 v2 v3 v4 v5 v6

Delete the vertex whose in-degree 0. (v7)

Topological Sort: Using in-degree

T: v1 v2 v3 v4 v5 v6 v7

Delete the vertex whose in-degree 0.

Topological Sort: Using in-degree

Steps for finding the topological ordering of a DAG:

Step-1: Compute in-degree for each of the vertices present in the DAG and initialize the count of visited nodes as 0;

Step-2: Add all **vertices with in-degree equals 0** into a queue

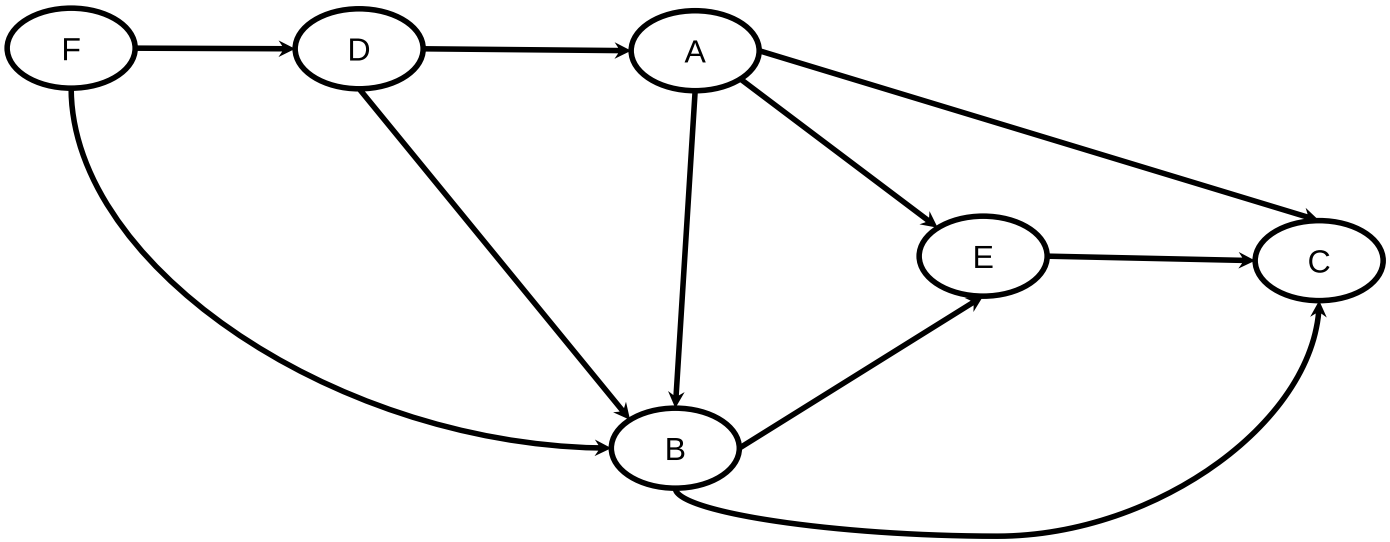
Step-3: Remove a vertex from the queue and then

- Increment count of visited nodes by 1;
- Decrease in-degree by 1 for all its neighboring nodes;
- If in-degree of a neighboring node is reduced to zero, then add it to the queue;

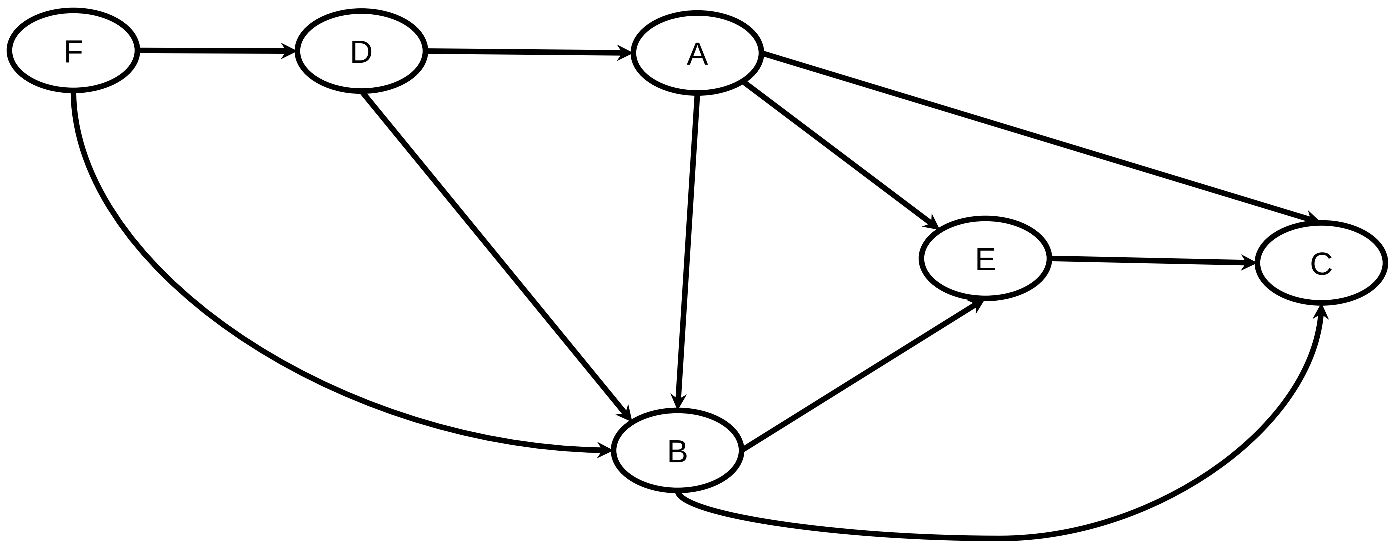
Step 4: Repeat Step 3 until **the queue is empty**;

Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph

Topological Sort

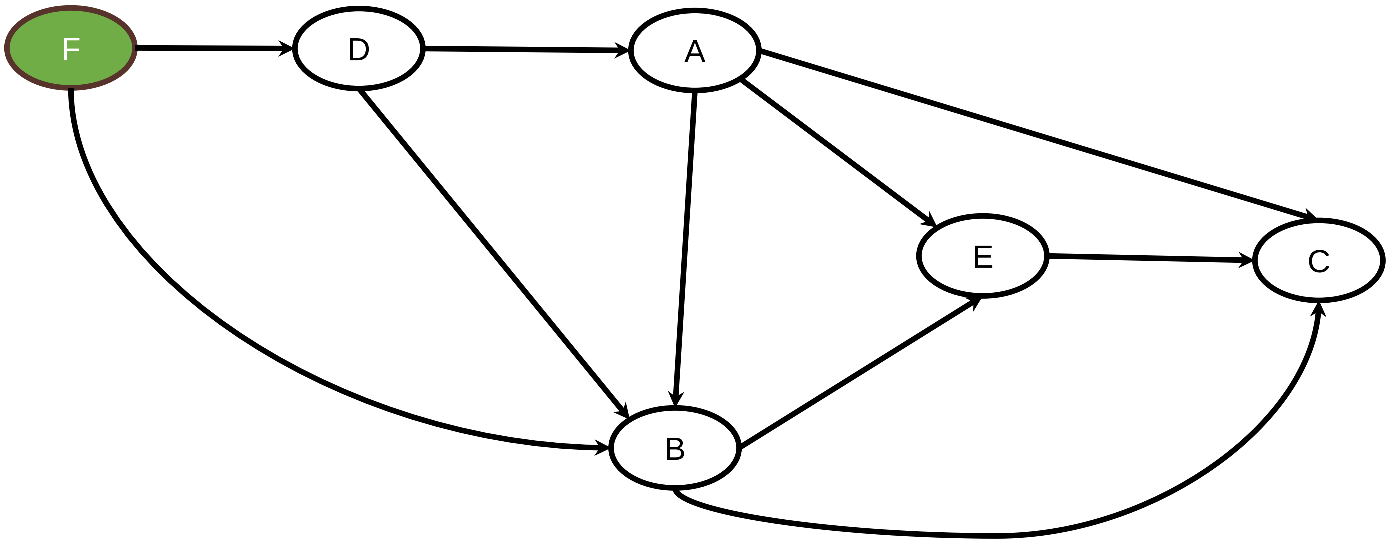


Topological Sort



In-degree	1	3	3	1	2	0
	A	B	C	D	E	F

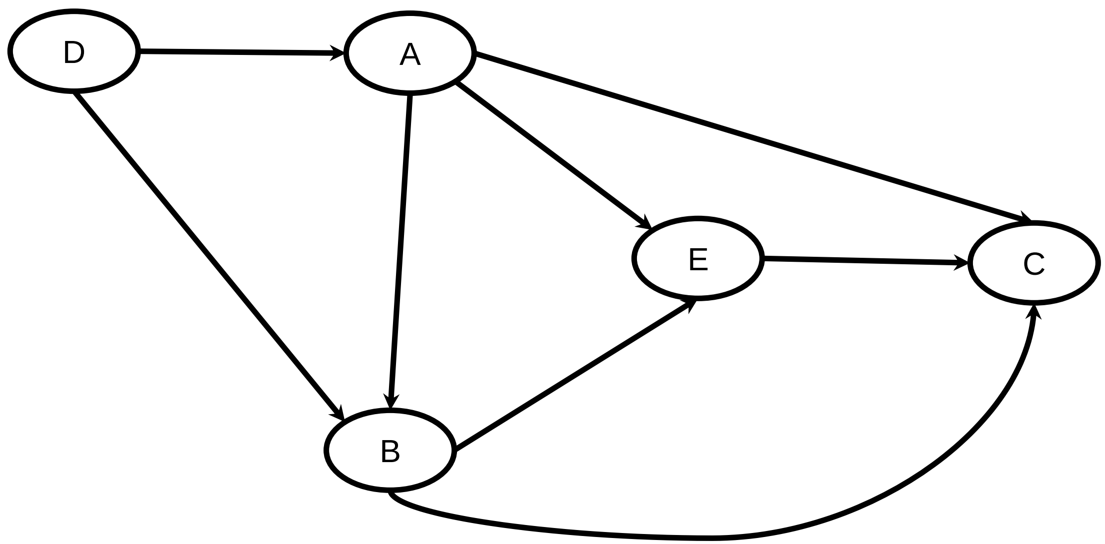
Topological Sort



In-degree	1	3	3	1	2	0
	A	B	C	D	E	F

Q: F

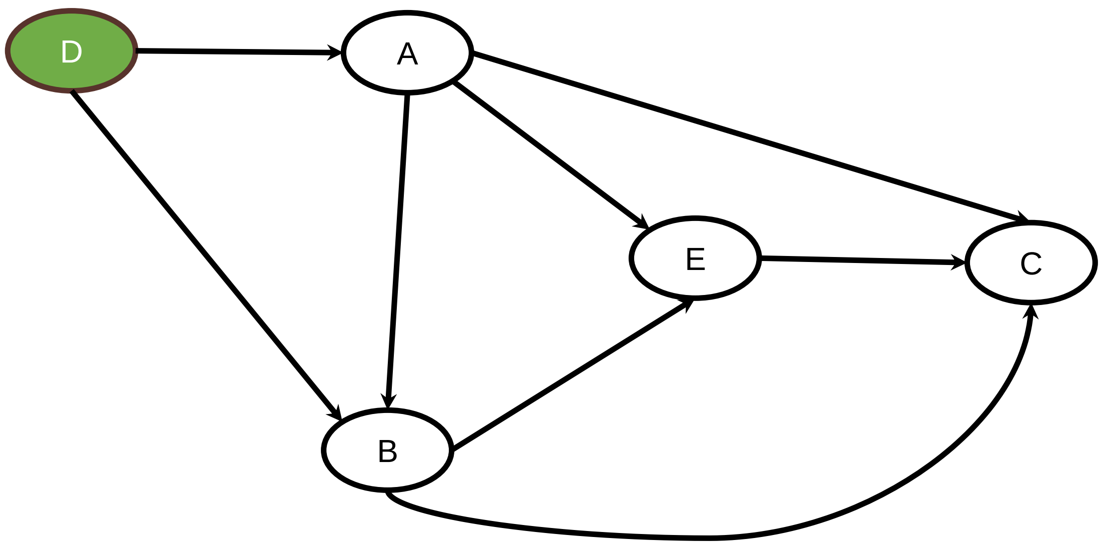
Topological Sort



In-degree	1	2	3	0	2	0
	A	B	C	D	E	F

Q: F

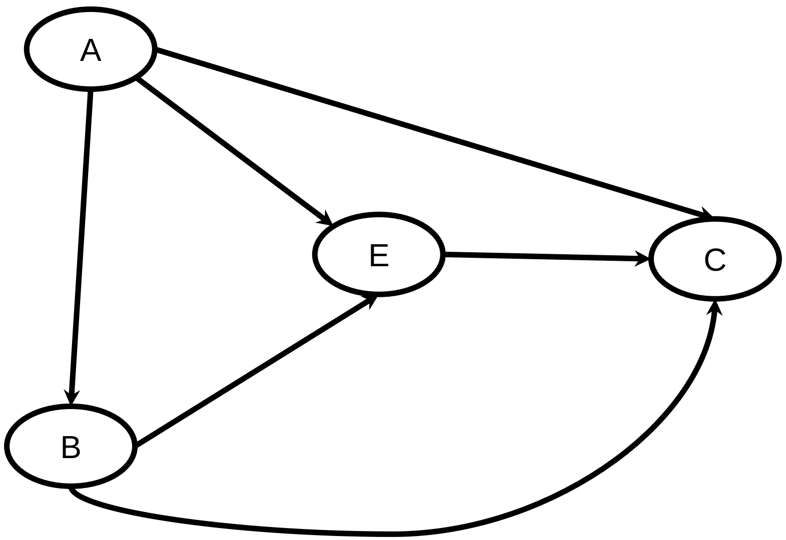
Topological Sort



In-degree	1	2	3	0	2	0
	A	B	C	D	E	F

Q: F D

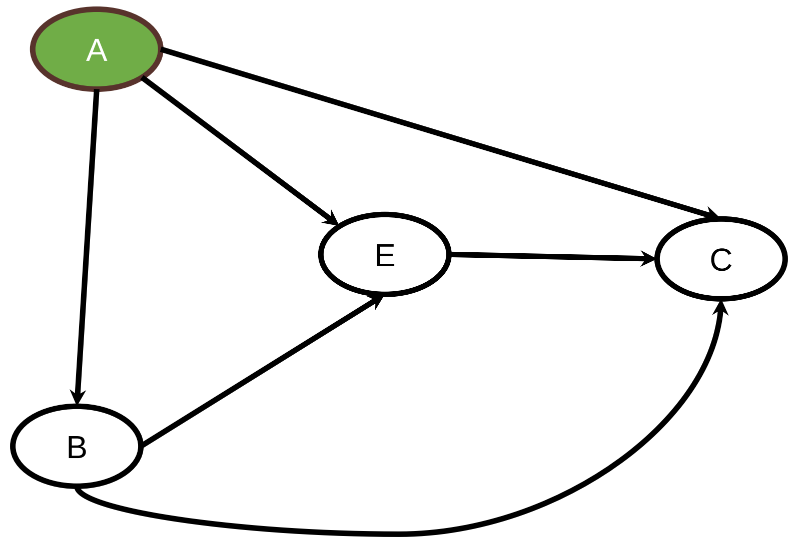
Topological Sort



In-degree	0	1	3	0	2	0
	A	B	C	D	E	F

Q: F D

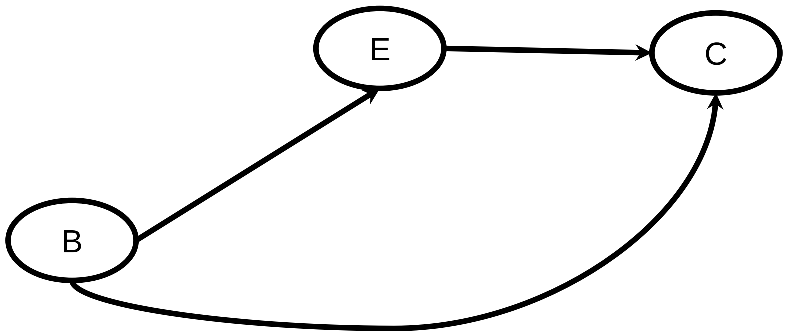
Topological Sort



In-degree	0	1	3	0	2	0
	A	B	C	D	E	F

Q: F D A

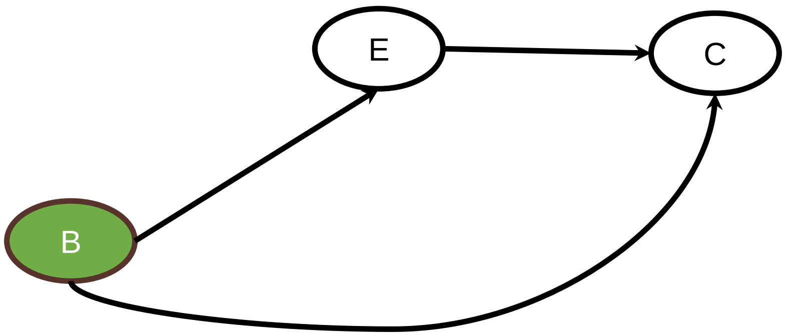
Topological Sort



In-degree	0	0	2	0	1	0
	A	B	C	D	E	F

Q: F D A

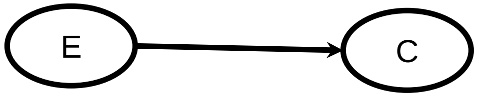
Topological Sort



In-degree	0	0	2	0	1	0
	A	B	C	D	E	F

Q: F D A B

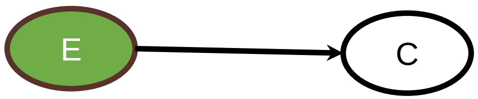
Topological Sort



In-degree	0	0	1	0	0	0
	A	B	C	D	E	F

Q: F D A B

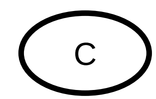
Topological Sort



In-degree	0	0	1	0	0	0
	A	B	C	D	E	F

Q: F D A B E

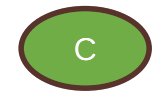
Topological Sort



In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E

Topological Sort



In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E C

Topological Sort

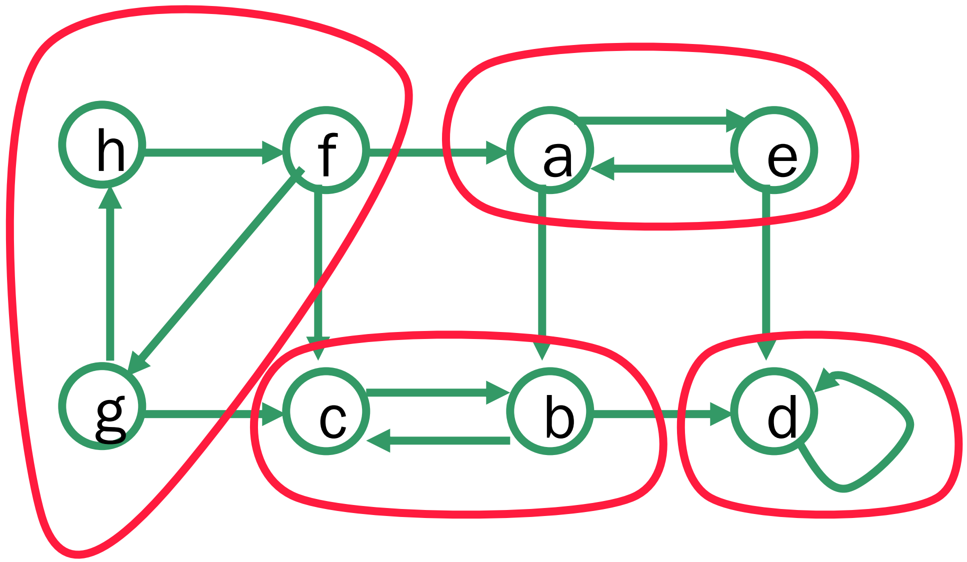
In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E C

DFS Application: Strongly Connected Components

- Consider a directed graph *G*.
- A strongly connected component (SCC) of the graph *G* is a maximal set of vertices with a (directed) path between every pair of vertices
- Problem: Find all the SCCs of the graph.

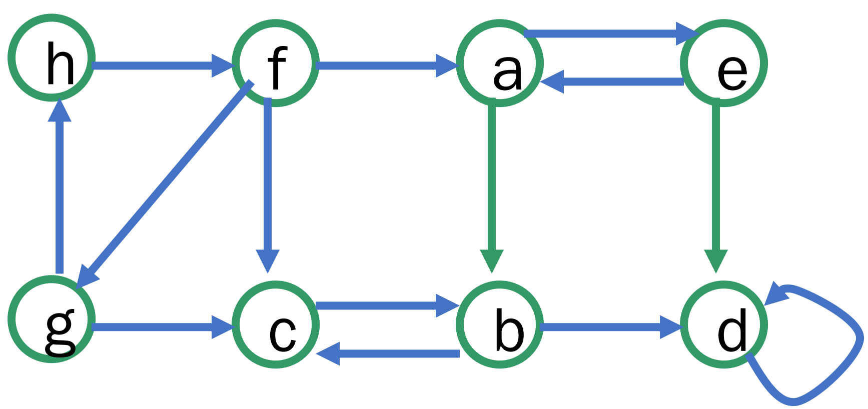
SCC Example



four SCCs

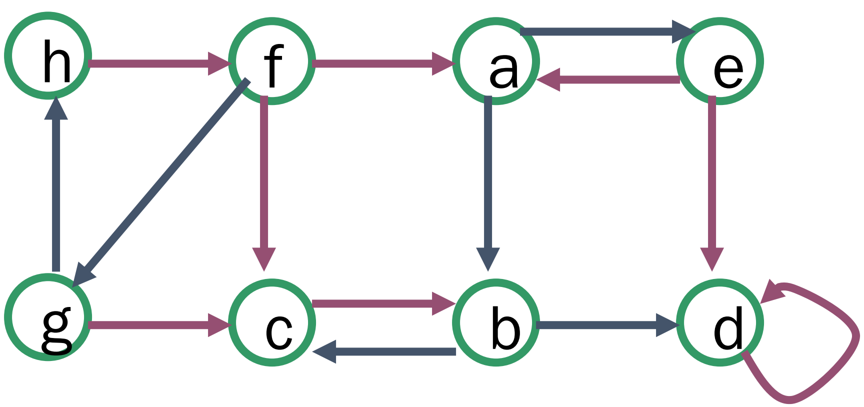
How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All vertices in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
    - Example: start DFS from the vertex *a* in the following graph



How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All vertices in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
  - Example: start DFS from the vertex *a* in the following graph



34

Main Idea of SCC Algorithm

- DFS tells us which vertices are reachable from the roots of the individual trees
- Also need information in the "other direction": is the root reachable from its descendants?
- Run DFS again on the "transpose" graph (reverse the directions of the edges)

35

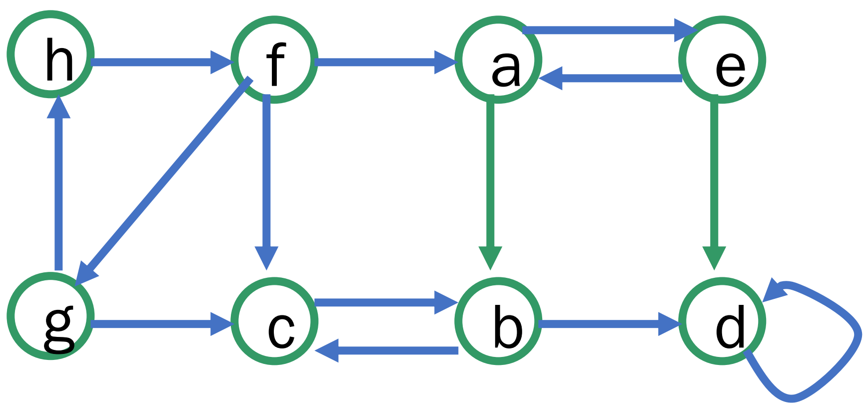
SCC Algorithm

input: directed graph G = (V, E)

1. call DFS(G) to compute finishing times
2. compute G^T // transpose graph
3. call DFS(G^T), considering vertices in decreasing order of finishing times
4. each tree from Step 3 is a separate SCC of G

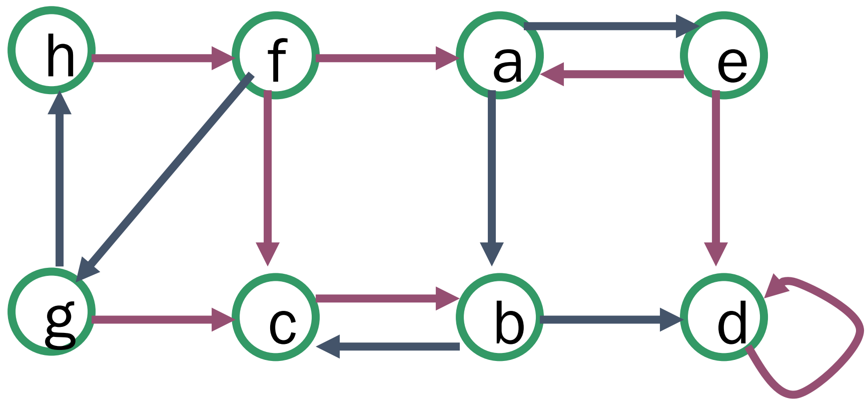
36

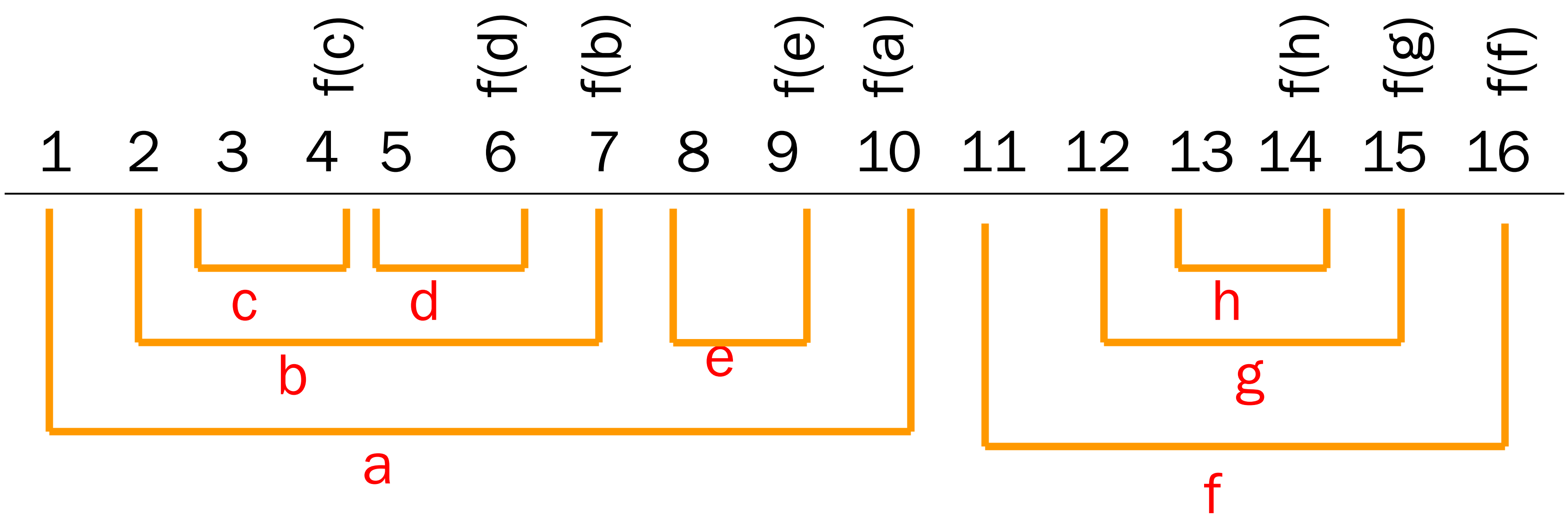
SCC Algorithm Example



input graph - run DFS

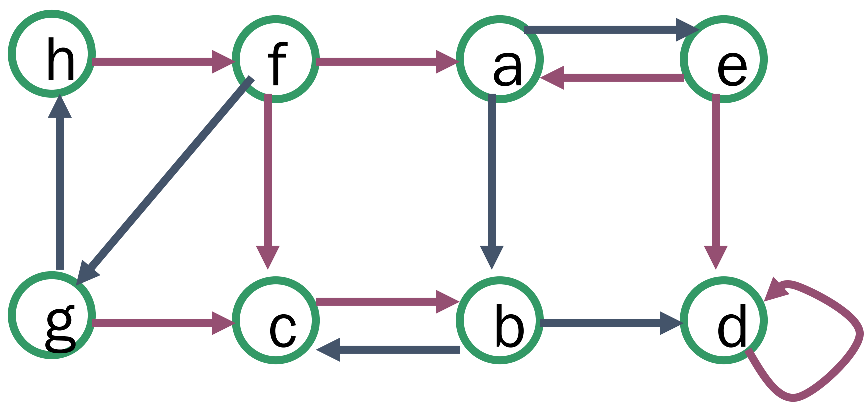
SCC Algorithm Example: After Step 1



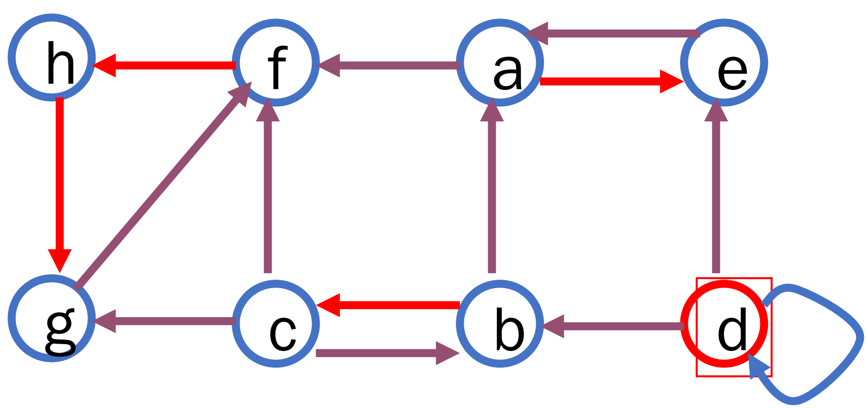


Order of vertices for Step 3: f, g, h, a, e, b, d, c

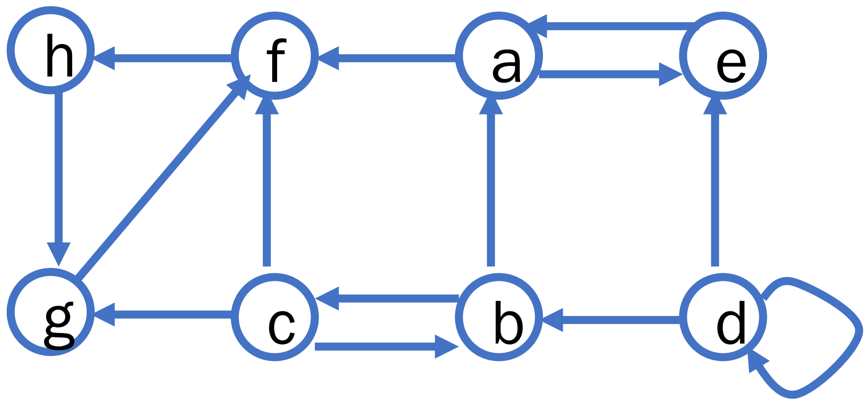
SCC Algorithm Example: After Step 1



Order of vertices for Step 3: f, g, h, a, e, b, d, c



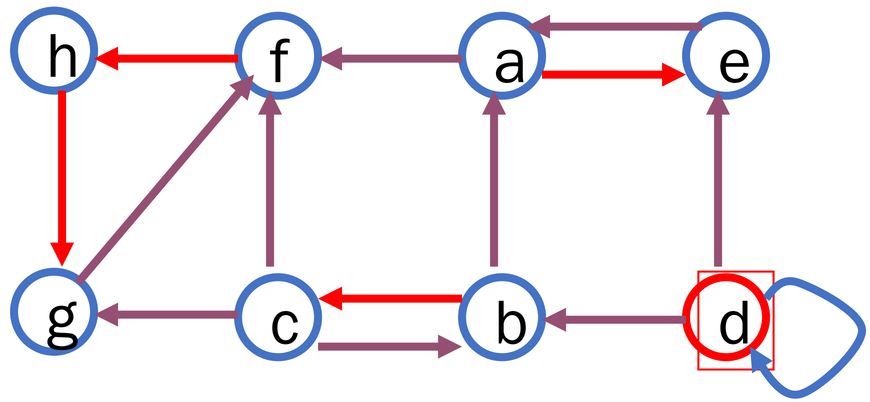
SCC Algorithm Example: After Step 2

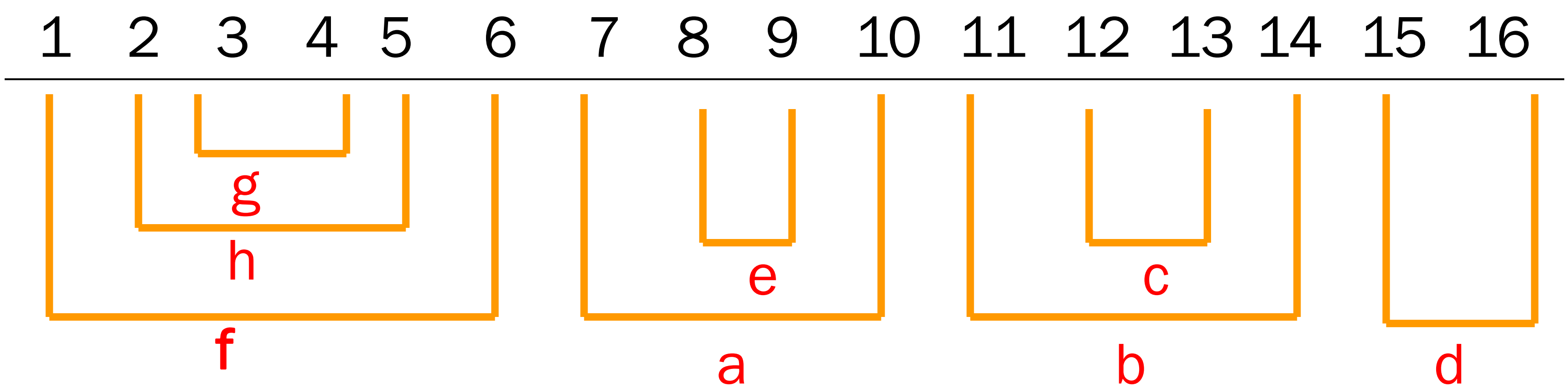


transposed input graph - run DFS with specified order of vertices

SCC Algorithm Example: After Step 3

Order of vertices for Step 3: f, g, h, a, e, b, d, c





SCCs are {f,h,g}, {a,e}, {b,c}, and {d}

Running Time of SCC Algorithm

- Step 1: $O(V + E)$ to run DFS
- Step 2: $O(V + E)$ to construct transpose graph, assuming adjacency list rep.
- Step 3: $O(V + E)$ to run DFS again
- Step 4: $O(V)$ to output result
- Total: $O(V + E)$

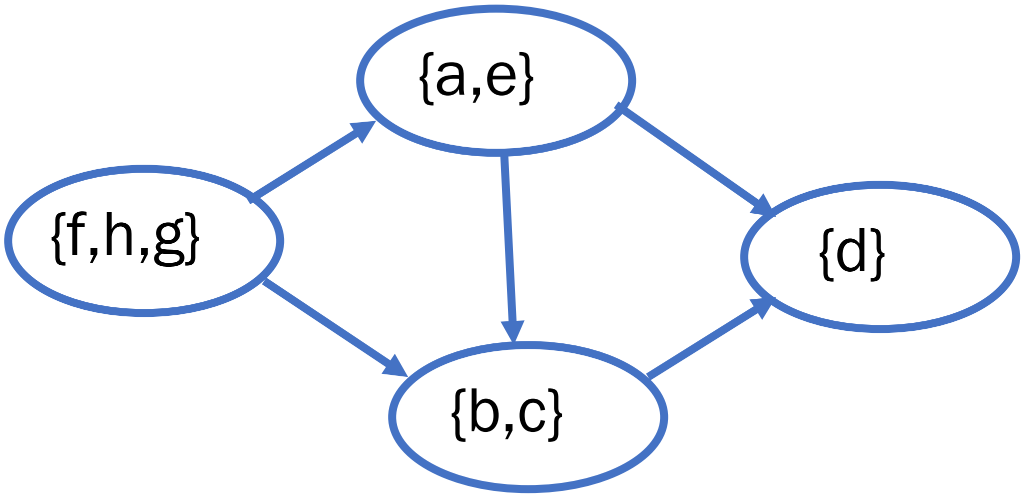
42

Correctness of SCC Algorithm

- Proof uses concept of component graph G^{SCC}, of G.
- Vertices are the SCCs of G; call them $C_1, C_2, \ldots, C_k$
- Put an edge from C_i to C_j iff G has an edge from a vertex in C_i to a vertex in C_j

43

Example of Component Graph



based on example graph from before

Facts About Component Graph

- Claim: G^{SCC} is a directed acyclic graph.
- Why?
- Suppose there is a cycle in G^{SCC} such that component C_i is reachable from component C_j and vice versa.
- Then C_i and C_j would not be separate SCCs.

45

Facts About Component Graph

- Consider any component C during Step 1 (running DFS on G)
- Let d(C) be *earliest* discovery time of any vertex in C
- Let f(C) be *latest* finishing time of any vertex in C
- Lemma: If there is an edge in G^{SCC} from component C' to component C, then

$$f(C') > f(C).$$

Proof of Lemma

- Case 1: d(C') < d(C).
- Suppose x is first vertex discovered in C'.
- By the way DFS works, all vertices in C' and C become descendants of x.
- Then x is last vertex in C' to finish and finishes after all vertices in C.
- Thus f(C') > f(C).

Proof of Lemma

- Case 2: $d(C') > d(C)$.
- Suppose y is first vertex discovered in C.
- By the way DFS works, all vertices in C become descendants of y.
- Then y is last vertex in C to finish.
- Since $C' \to C$, no vertex in C' is reachable from y, so y finishes before any vertex in C' is discovered.
- Thus $f(C') > f(C)$.

48

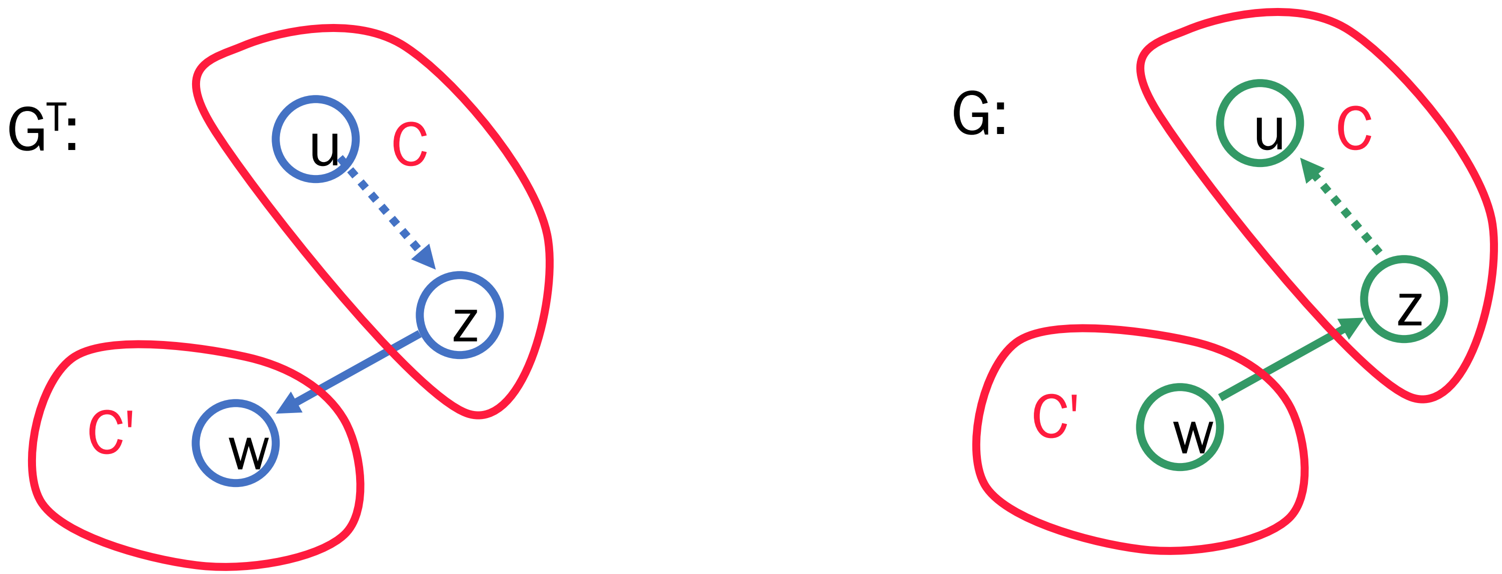
SCC Algorithm is Correct

- Prove this theorem by induction on number of trees found in Step 3 (running DFS on G^T).
- Hypothesis is that the first k trees found constitute k SCCs of G.
- Basis: k = 0. No work to do !

- Induction: Assume the first k trees constructed in Step 3 (running DFS on G^T) correspond to k SCCs; consider the (k+1)st tree.
- Let u be the root of the (k+1)st tree.
- u is part of some SCC, call it C.
- By the inductive hypothesis, C is not one of the k SCCs already found and all so vertices in C are unvisited when u is discovered.
  - By the way DFS works, all vertices in C become part of u's tree

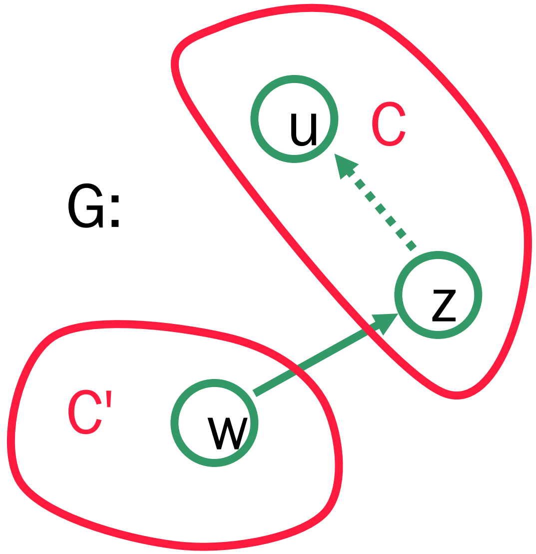
SCC Algorithm is Correct

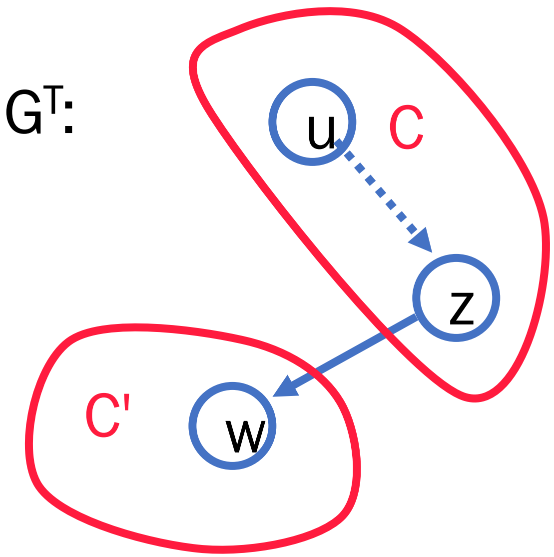
- Show *only* vertices in C become part of u's tree. Consider an outgoing edge from C.



SCC Algorithm is Correct

- By lemma, in Step 1 (running DFS on G) the last vertex in C' finishes after the last vertex in C finishes [f(C') > f(C)].
- Thus in Step 3 (running DFS on G^T), some vertex in C' is discovered before any vertex in C is discovered.
- Thus in Step 3, all of C', including w, is already visited before u's DFS tree starts





Thank you