CSC 226

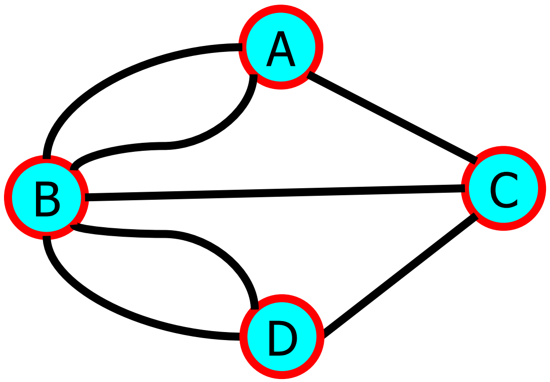
Algorithms and Data Structures: II

Rich Little
rlittle@uvic.ca

Abstract Meaning of the Term Graph

- A ***graph*** $G = (V, E)$ is a set V of ***vertices*** (*nodes*) and a collection E of pairs from V, called ***edges*** (*arcs*).

- **Graph Example:**

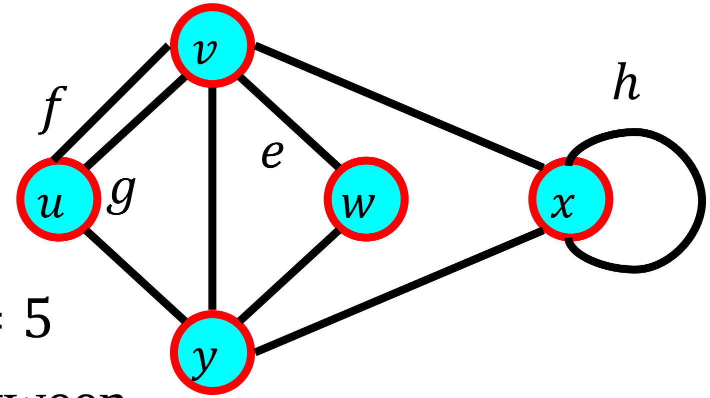


$$V = \{A, B, C, D\}$$

$$E = \left\{ \begin{array}{l} \{A, B\}, \{A, B\}, \{A, C\}, \\ \{B, C\}, \{B, D\}, \{B, D\}, \{C, D\} \end{array} \right\}$$

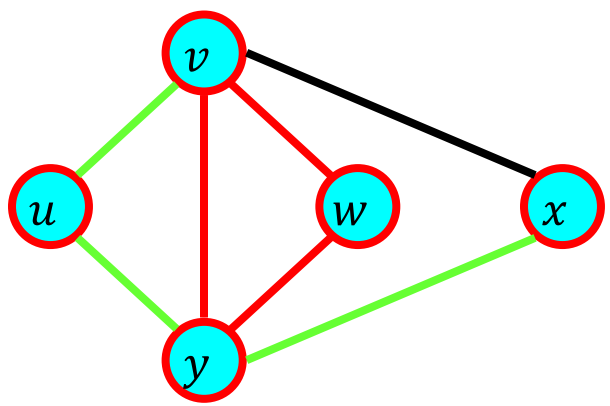
Undirected Edges

- An *undirected edge e* represents a *symmetric* relation between two vertices *v* and *w* represented by the vertices.
  - We usually write $e = \{v, w\}$, where $\{v, w\}$ is an unordered pair.
  - *v*, *w* are the *endpoints* of the edge
  - *v* is *adjacent* to *w*
  - *e* is *incident* upon *v* and *w*
  - The *degree* of a vertex is the number of incident edges, eg. $\deg(v) = 5$
  - *parallel* edges – more than one edge between a pair of vertices, eg. f and g
  - *self-loop* – edge that connects a vertex to itself, eg. *h*
  - Typically, the number of vertices is denoted by *n* and the number of edges by *m*.



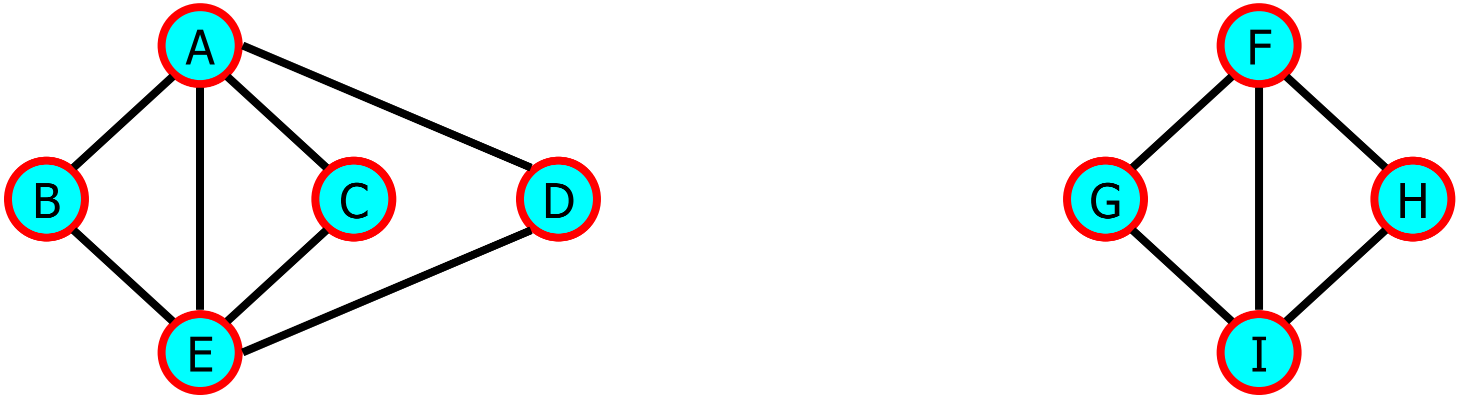
Undirected Paths

- A ***walk*** in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that there exist edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, …, $\{v_{n-1}, v_n\}$
- The ***length*** of a walk is the number of edges
  - if $v_1 = v_n$, ***closed***, otherwise ***open***
- If no edge is repeated, it's a ***trail***
- A closed trail is a ***circuit***
- If no vertex is repeated, it's a ***path***
- A ***cycle*** is a path with the same start and end vertices



Connected Graphs

- A graph is ***connected*** if every pair of vertices is connected by a path.
- Example
  - ➢Two *connected components* of a graph
  - ➢*Unconnected* graph



Simple Graphs

- A ***simple graph*** is a graph with no self-loops and no parallel or multi-edges

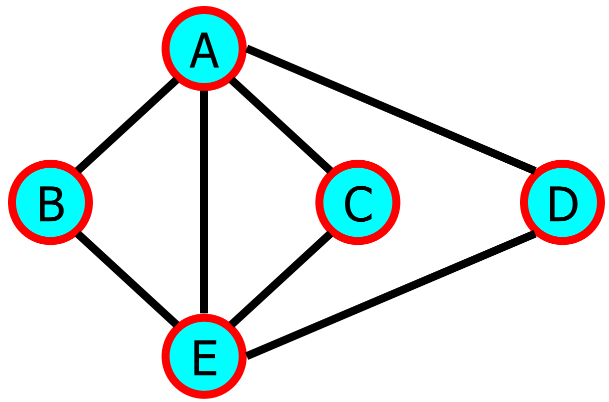
- **Theorem:** If $G = (V, E)$ is a graph with m edges, then

$$\sum_{v \in V} \deg(v) = 2m$$

- **Theorem:** Let G be a simple graph with n vertices and m edges. Then,

$$m \leq \frac{n(n-1)}{2}$$

- **Corollary:** A simple graph with n vertices has $O(n^2)$ edges.

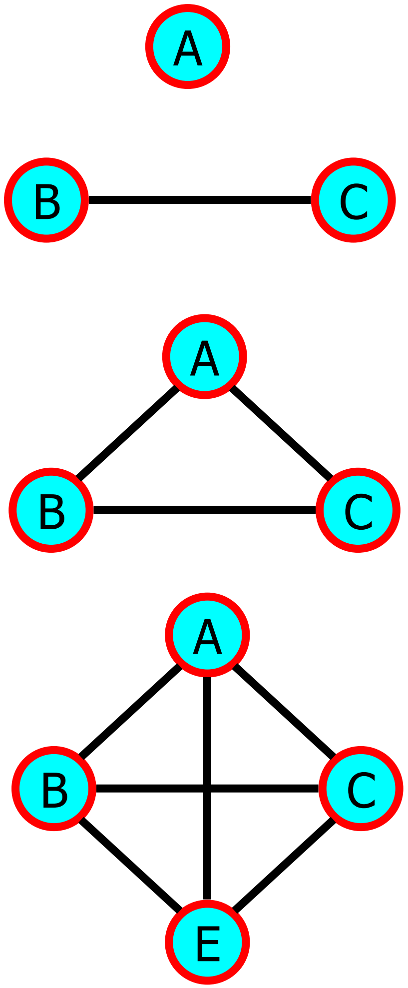


Complete Graphs

- A ***complete graph*** is a simple graph where an edge connects every pair of vertices

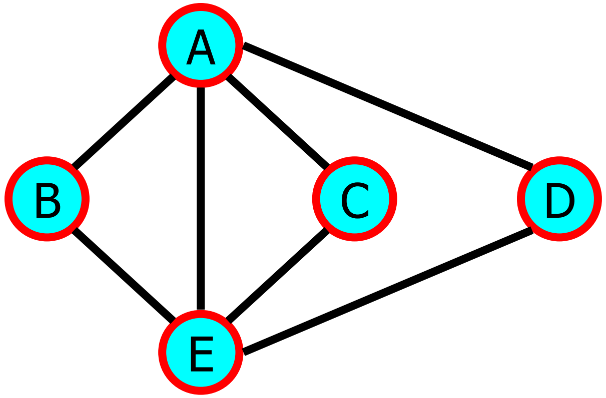
- The complete graph on n vertices has exactly $n(n - 1) / 2$ edges

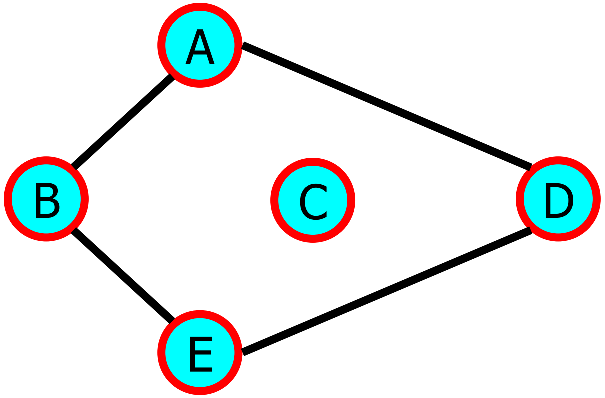
- A complete graph with at most one self loop per vertex on n vertices has exactly $n(n + 1) / 2$ edges



Subgraphs

- A ***subgraph*** of $G = (V, E)$ *is a graph* $G' = (V', E')$ where
  - V' is a subset of V
  - E' consists of edges $\{v, w\}$ in E such that both v and w are in V'
- A ***spanning subgraph*** of G contains all the vertices of G

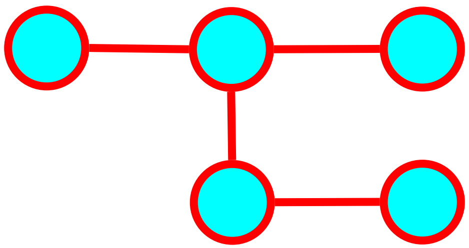




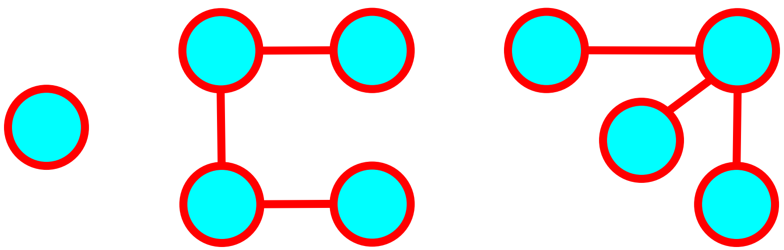
Trees and Forests

- A (*free*) *tree* is an undirected graph T such that
  - ➢ T is connected
  - ➢ T has no cycles

  This definition of tree is different from the one of a rooted tree
- A *forest* is an undirected graph without cycles
- The connected components of a forest are trees



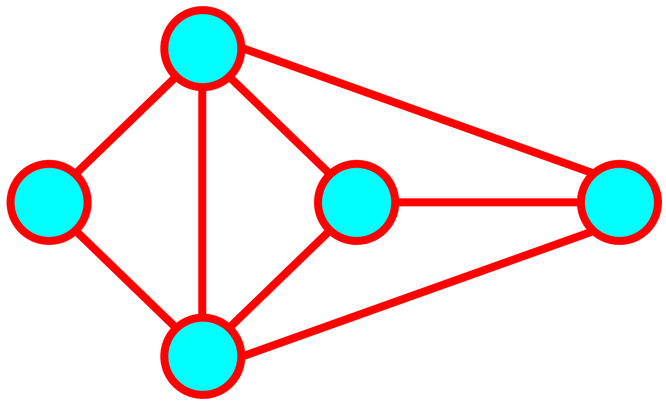
Tree



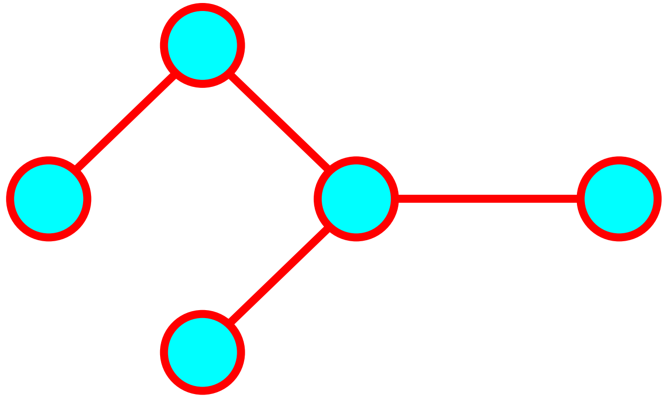
Forest

Spanning Trees and Forests

- A *spanning tree* of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A *spanning forest* of a graph is a spanning subgraph that is a forest



Graph



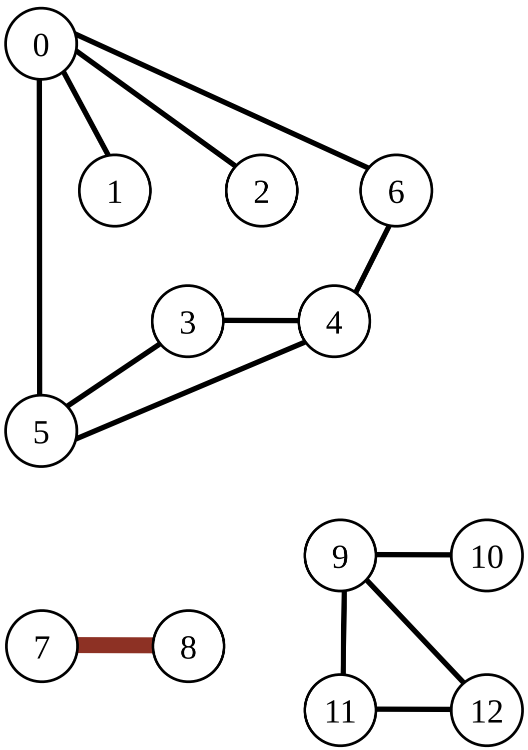
Spanning tree

Properties of Trees, Forests and Graphs

- Theorem: Let G be an undirected simple graph with n vertices and m edges. Then we have the following:
  - If G is connected, then $m \geq n - 1$.
  - If G is a tree, the $m = n - 1$.
  - If G is a forest, then $m \leq n - 1$.

Graph representation: set of edges

- Maintain a list of the edges (linked list or array).

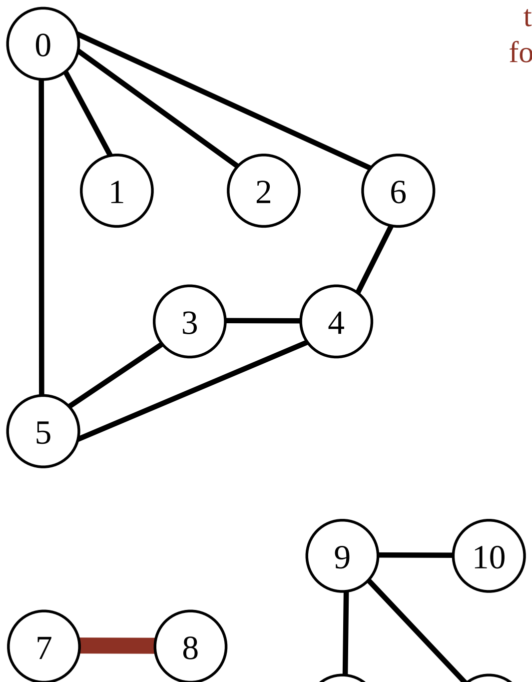


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Q. How long to iterate over vertices adjacent to *v* ?

Graph representation: adjacency matrix

- Maintain a two-dimensional *n*-by-*n* boolean array;
  for each edge *v*–*w* in graph: adj[v][w] = adj[w][v] = true.



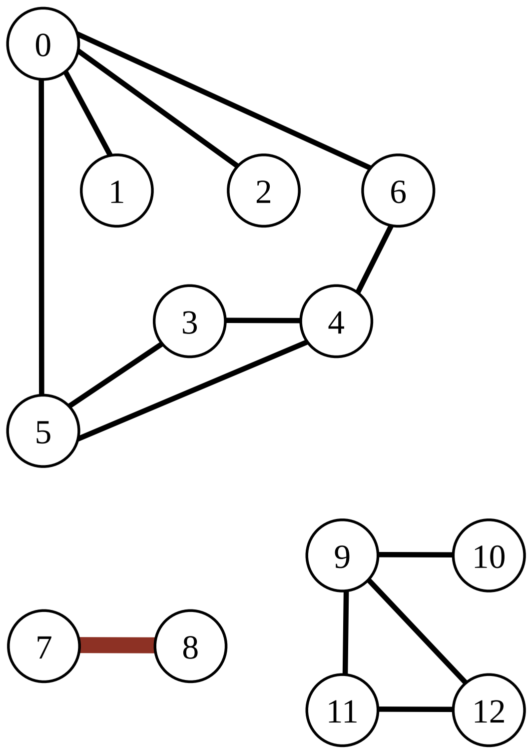
two entries for each edge

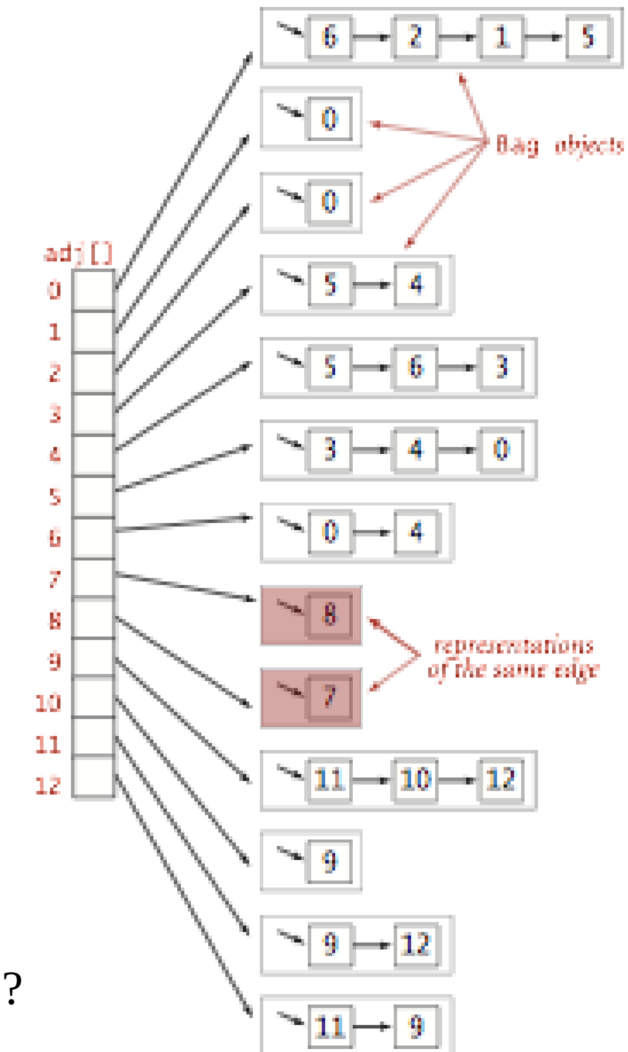
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0

Q. How long to iterate over vertices adjacent to *v* ?

Graph representation: adjacency lists

- Maintain vertex-indexed array of lists.





Q. How long to iterate over vertices adjacent to v ?

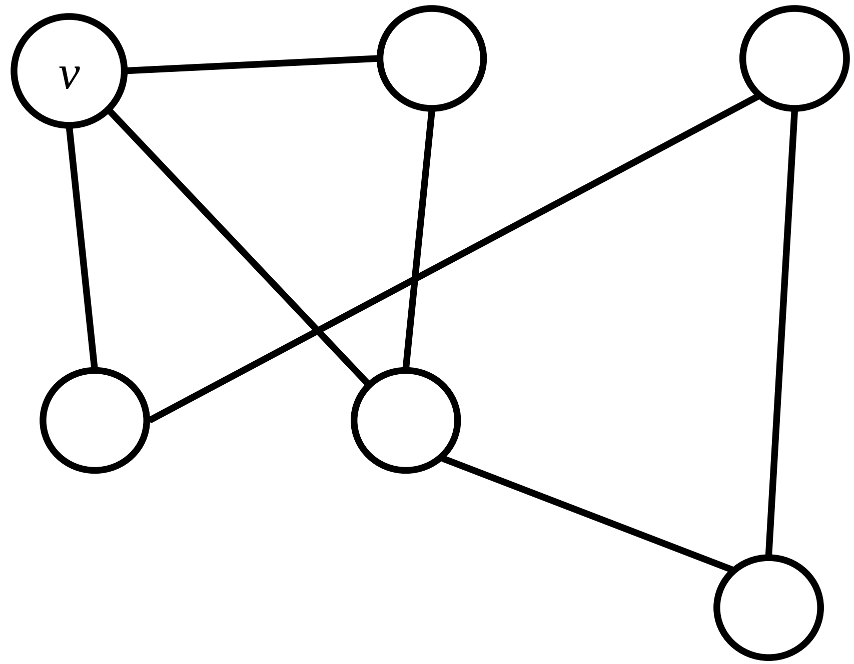
Algorithm DFS

Algorithm DFS(*G*, *v*):

Input: A graph *G* and a vertex *v* of *G*

*Output***:** A labeling of the edges in the connected component as discovery edges and back edges

Label *v* as explored
for each edge, *e*, incident to *v* **do**
  if *e* is unexplored **then**
    Let *w* be opposite node of *e*
    if *w* is unexplored **then**
      Label *e* as *discovery* edge
      DFS(G,*w*)
    else
      Label *e* as *back* edge



Algorithm BFS

Algorithm *BFS*(*G*, *s*):

Input: A graph *G* and a vertex *s* of *G*

***Output*:** A labeling of the edges as *discovery* edges and *cross* edges

Q ← new empty queue

Label *s* as explored

Q.enqueuer(*s*)

while *Q* is not empty **do**

v ← *Q.dequeue*()

for each edge, *e* = {*v*,*w*}, incident on *v* **do**

if *e* is unexplored **then**

if *w* is unexplored **then**

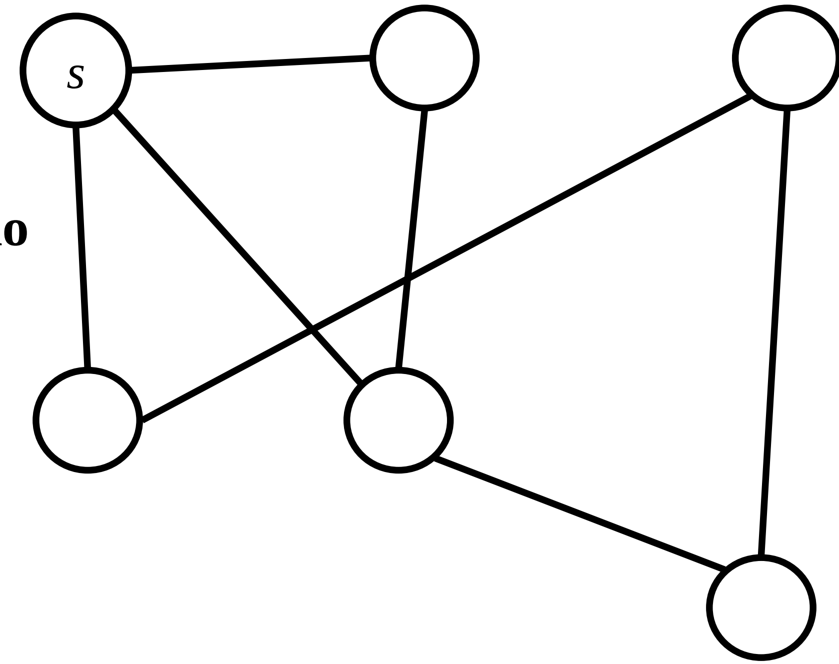
Label *e* as a *discovery* edge

Mark *w* as explored

Q.enqueue(*w*)

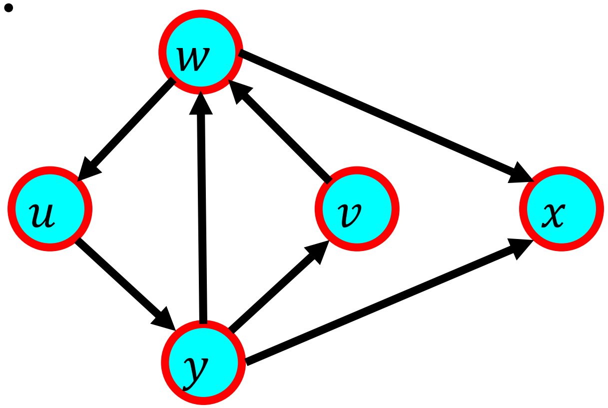
else

Label *e* as a *cross* edge



Directed Edges or Arcs

- A *directed edge* (*or arc*) *e* represents an *asymmetric* relation between two vertices *v* and *w*.
  e = (*v*, w) denotes an ordered pair.
  - *v*, *w* are the endpoints of the edge
  - *v* is *adjacent* to *w*
  - *e* is *incident* upon *v* and *w*
  - The arc goes from the *source* vertex *v* to the *destination* vertex *w*



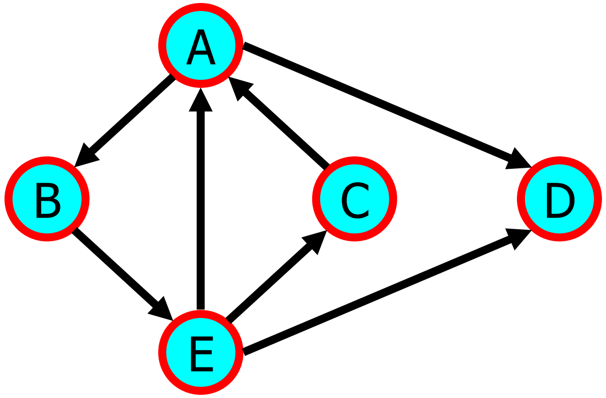
- The *indegree* of a vertex is the number of incoming arcs
- The *outdegree* of a vertex is the number of outgoing arcs

Simple Graphs

- A ***simple digraph*** is a graph with no self-loops and no parallel or multi-edges

- **Theorem:** If $G = (V, E)$ is a digraph with m edges, then

$$\sum_{v \in V} \operatorname{indeg}(v) = \sum_{v \in V} \operatorname{outdeg}(v) = m$$



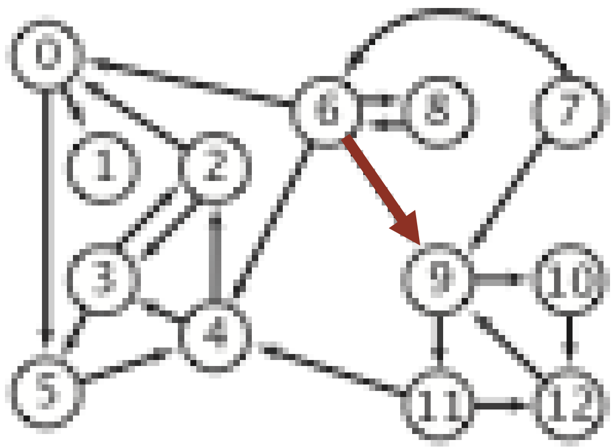
- **Theorem:** Let G be a simple digraph with n vertices and m edges. Then,

$$m \leq n(n-1)$$

- **Corollary:** A simple digraph with n vertices has $O\left(n^2\right)$ edges.

Digraph representation: set of edges

- Store a list of the edges (linked list or array).

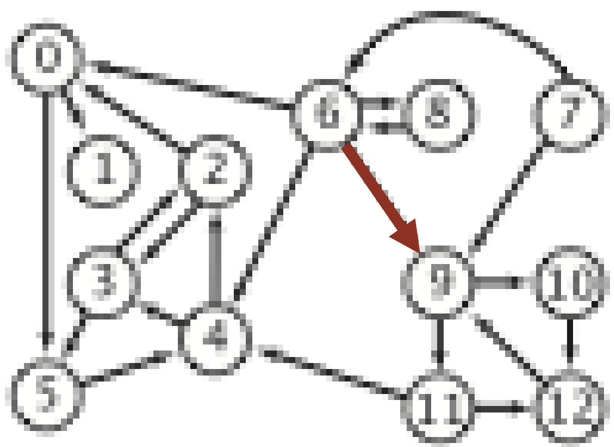


0	1
0	5
2	0
2	3
3	2
3	5
4	2
4	3
5	4
6	0
6	4
6	8
6	9
7	6
7	9
8	6
9	10
9	11
10	12
11	4
11	12
12	9

Digraph representation: adjacency matrix

- Maintain a two-dimensional V-by-V boolean array;
  for each edge $v \to w$ in the digraph: adj[v][w] = true.

Note: parallel edges disallowed



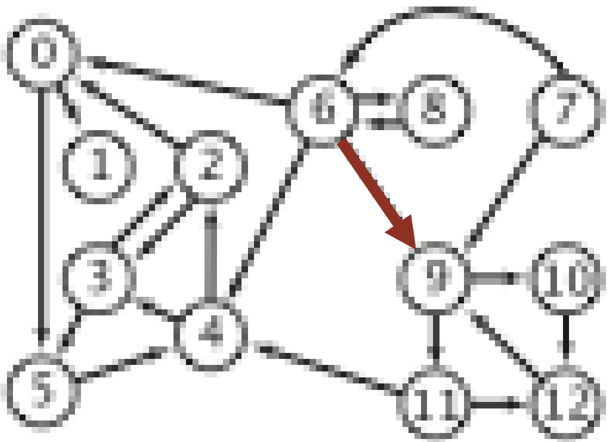
to

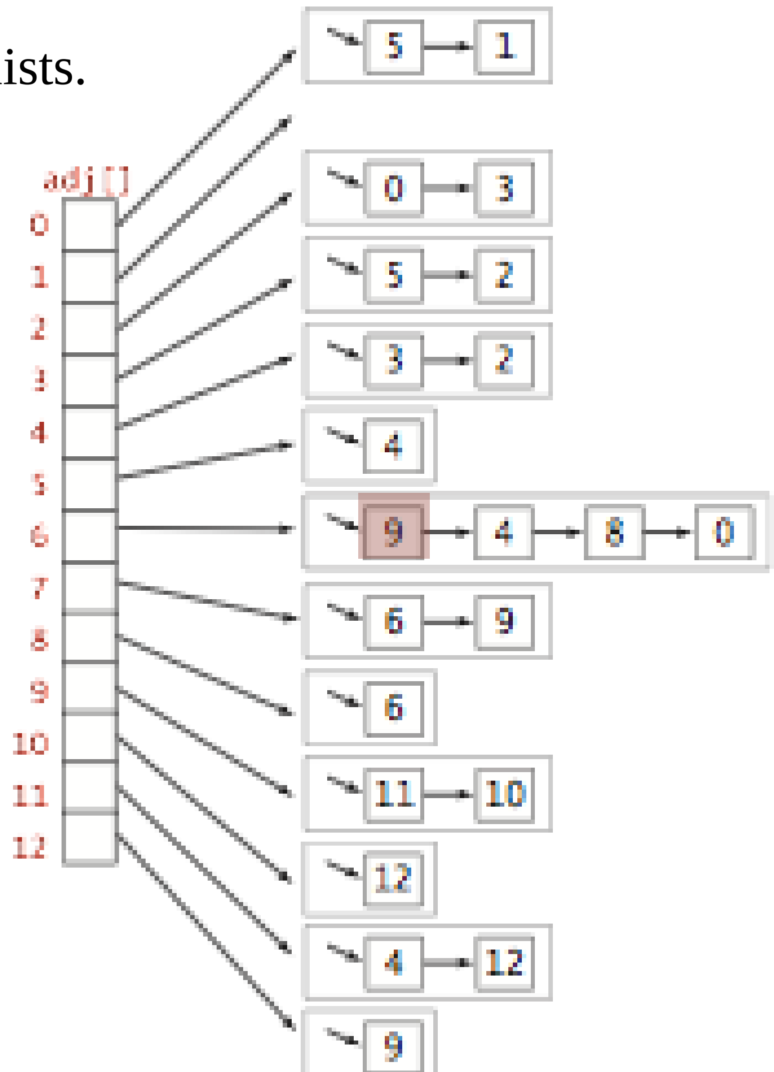
from

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1	1	0	0	0
7	0	0	0	0	0	0	1	0	0	1	0	0	0

Digraph representation: adjacency lists

- Maintain vertex-indexed array of lists.





Algorithm Directed DFS

Algorithm DirectedDFS(G, v):

Input: A digraph G and a vertex v of G

Output: A label of the edges as *discovery*, *back*, *forward* or *cross* edges

Label v as active

for each outgoing edge e **do**

    if e is unexplored **then**

        Let w be the destination of e

        if w is unexplored and not active **then**

            Label e as a *discovery* edge

            DirectedDFS(G,w)

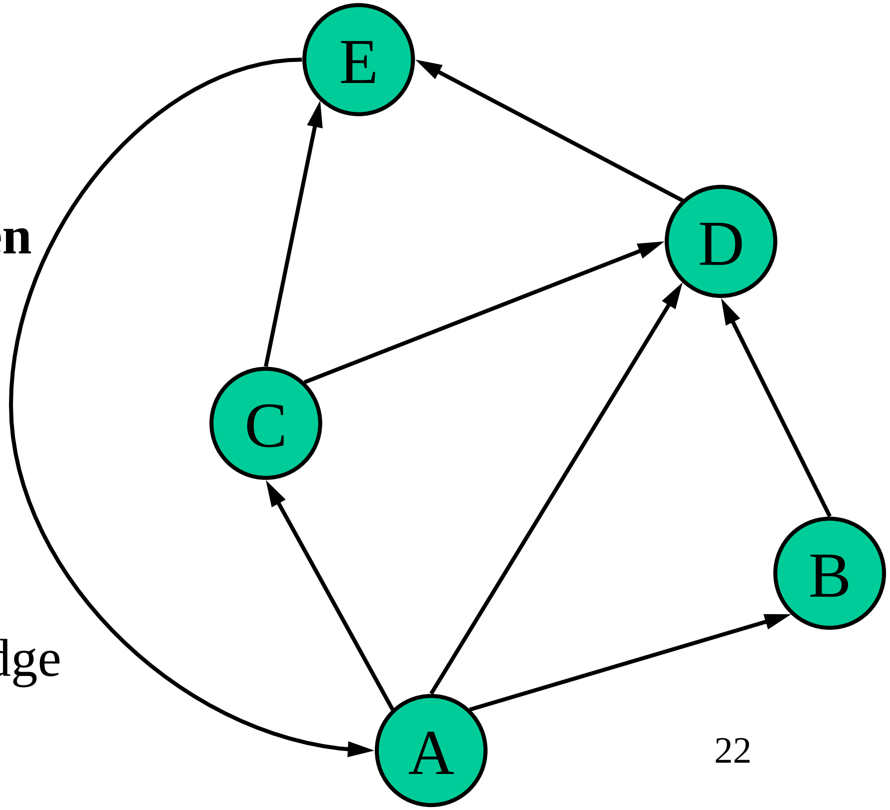
        else if w is active **then**

            mark edge e as a *back* edge

        else

            mark edge e as a *forward*/*cross* edge

Label v as explored



Algorithm Directed BFS

Algorithm *BFS*(*G*, *s*):

Input: A graph *G* and a vertex *s* of *G*

Output: A labeling of the edges as *discovery*, *back* or *cross* edges

$Q \leftarrow$ new empty queue

Label *s* as explored

Q.enqueuer(*s*)

while *Q* is not empty **do**

  $v \leftarrow$ *Q.dequeue*()

  for each outgoing edge, $e = (v,w)$, incident on *v* **do**

    if *e* is unexplored **then**

      if *w* is unexplored **then**

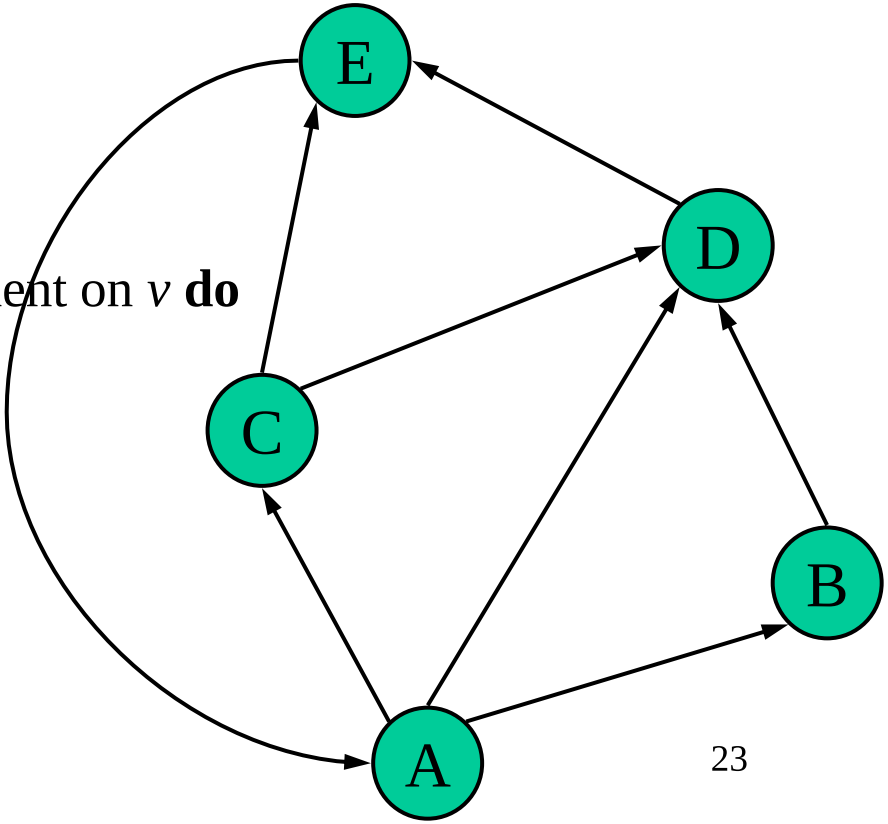
        Label *e* as a *discovery* edge

        Mark *w* as explored

        Q.enqueue(*w*)

    else

        Label *e* as a *back*/*cross* edge



Connected Digraphs

- Given vertices u and v of a digraph G, we say v is ***reachable*** from u if G has a directed path from u to v.

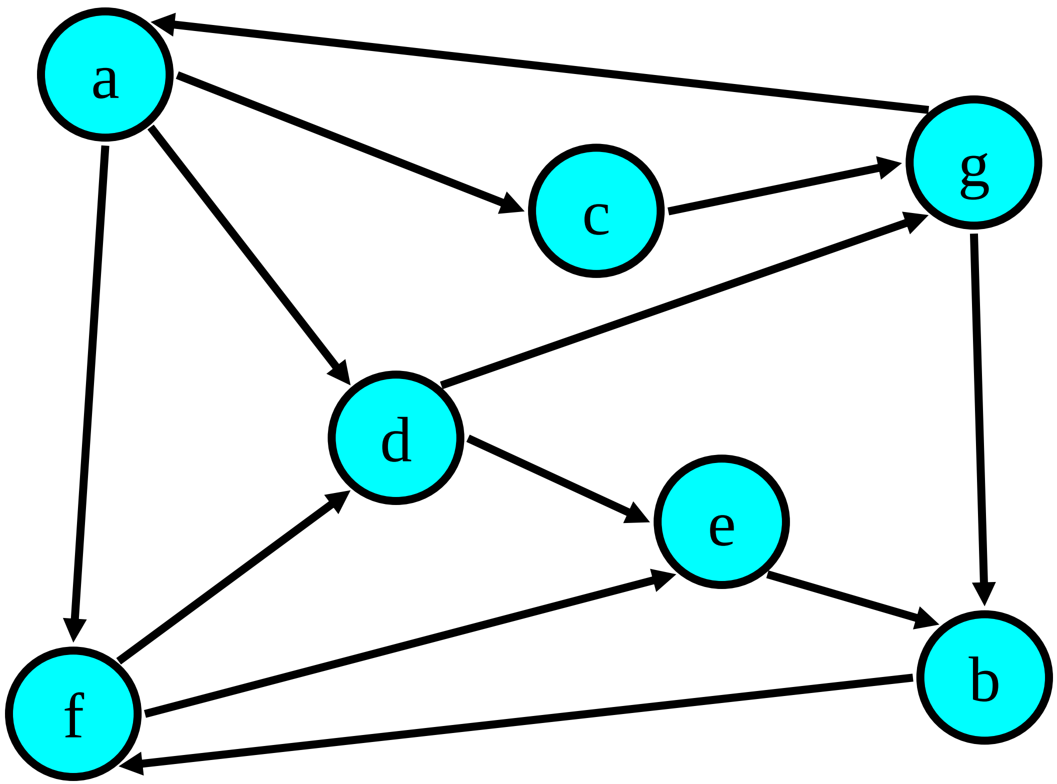
- A digraph G is ***connected*** if every pair of vertices is connected by an undirected path.

- A digraph G is ***strongly connected*** if for every pair of vertices u and v of G, u is reachable from v and v is reachable from u.

Strong Connectivity

- Each vertex can reach all other vertices

G:

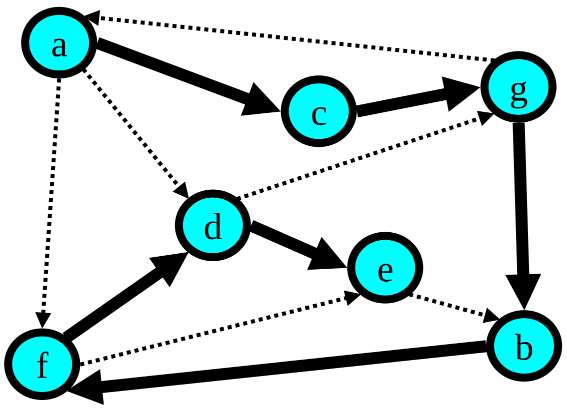


Strong Connectivity Algorithm

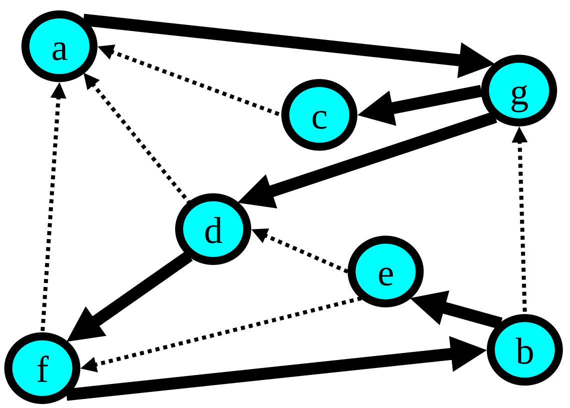
- Pick a vertex v in G.
- Perform a DFS from v in G.
  - ➢ If there's a w not visited, print "no".
- Let G' be G with edges reversed.
- Perform a DFS from v in G'.
  - ➢ If there's a w not visited, print "no".
  - ➢ Else, print "yes".

- Running time: $O(n + m)$.

G:

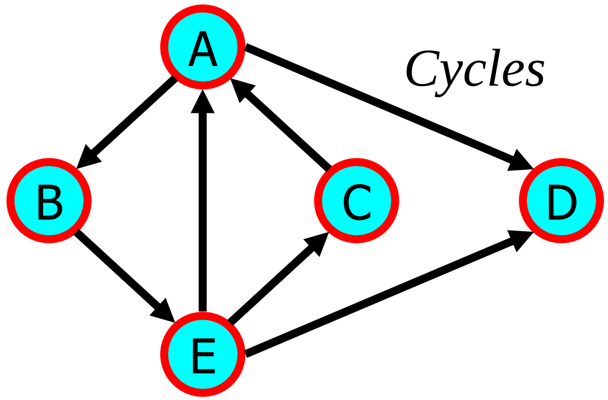


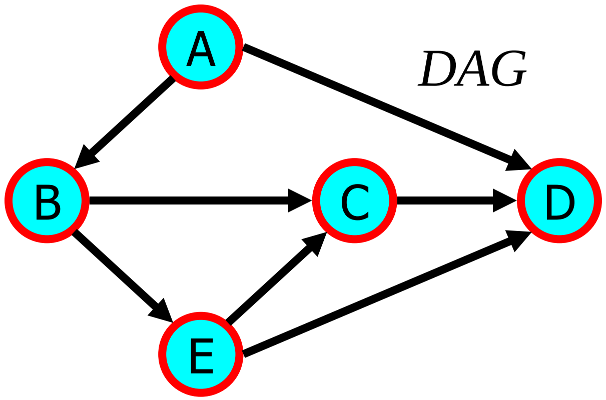
G′:



Directed Acyclic Graphs (DAGs)

- A ***directed acyclic graph*** (DAG) is a directed graph with no cycles.
- DAGs are more general than trees, but less general than arbitrary directed graphs.





DAGs and Topological Ordering

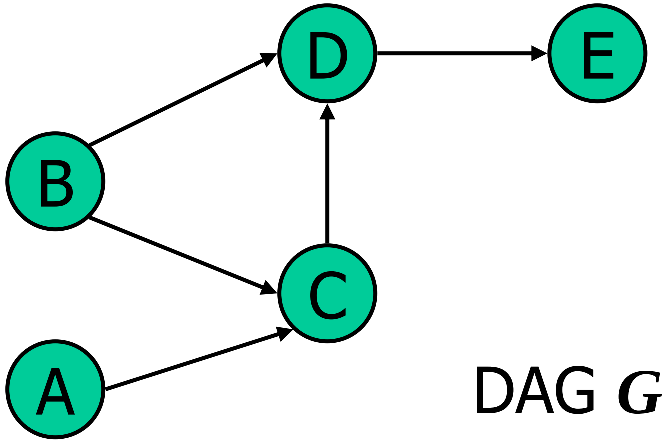
- A **topological ordering** of a digraph is a numbering

$$v_1, \ldots, v_n$$

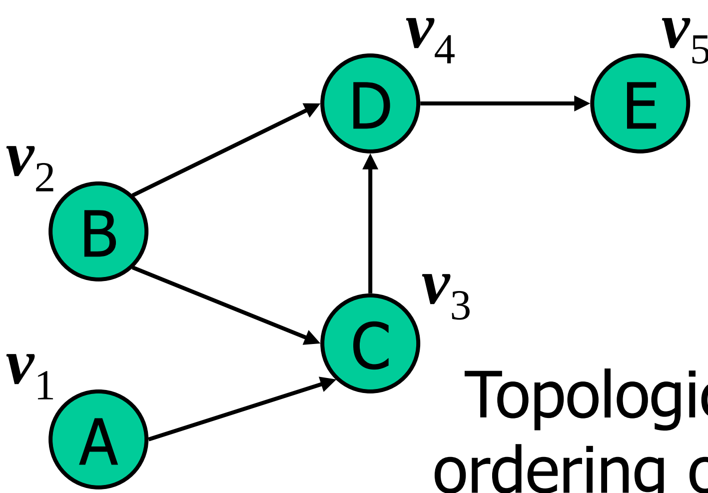
of the vertices such that for every edge (v_i, v_j), we have $i < j$

- **Theorem:**

A digraph admits a topological ordering if and only if it is a DAG



DAG G

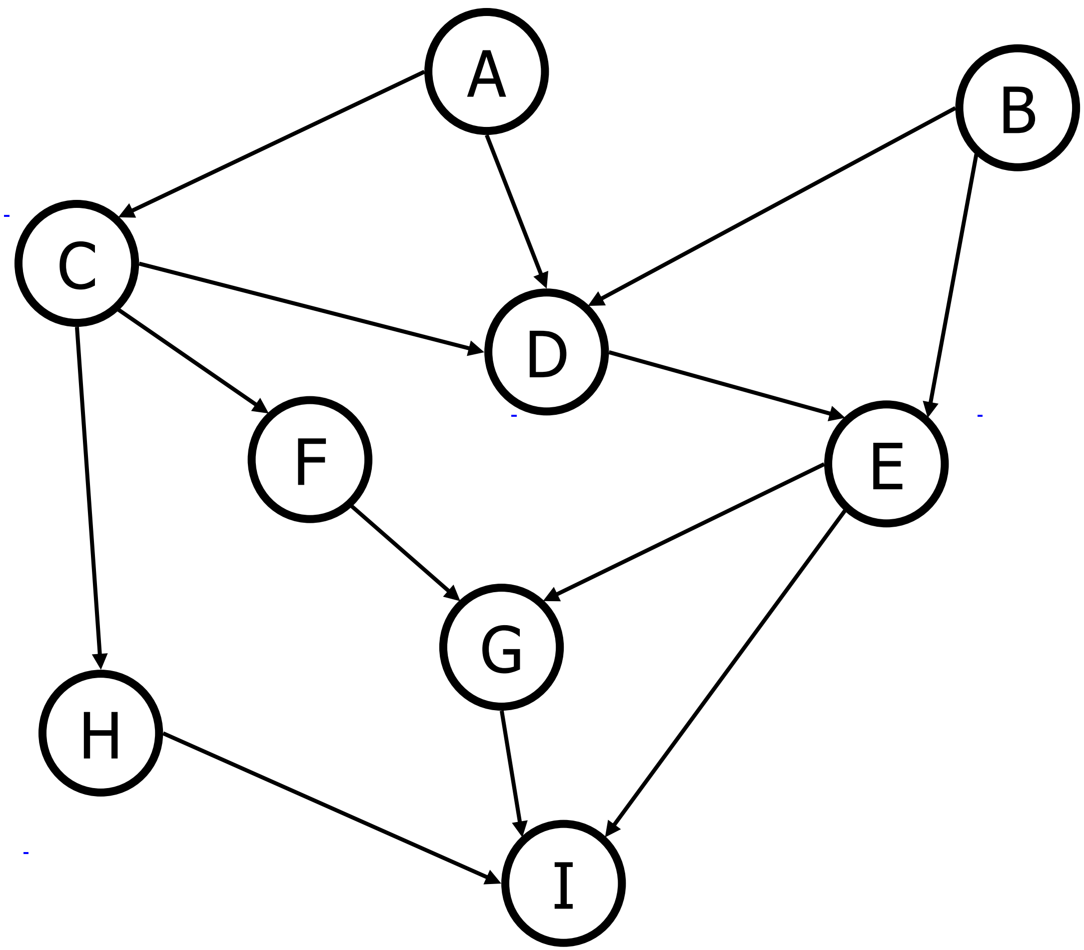


Topological ordering of G

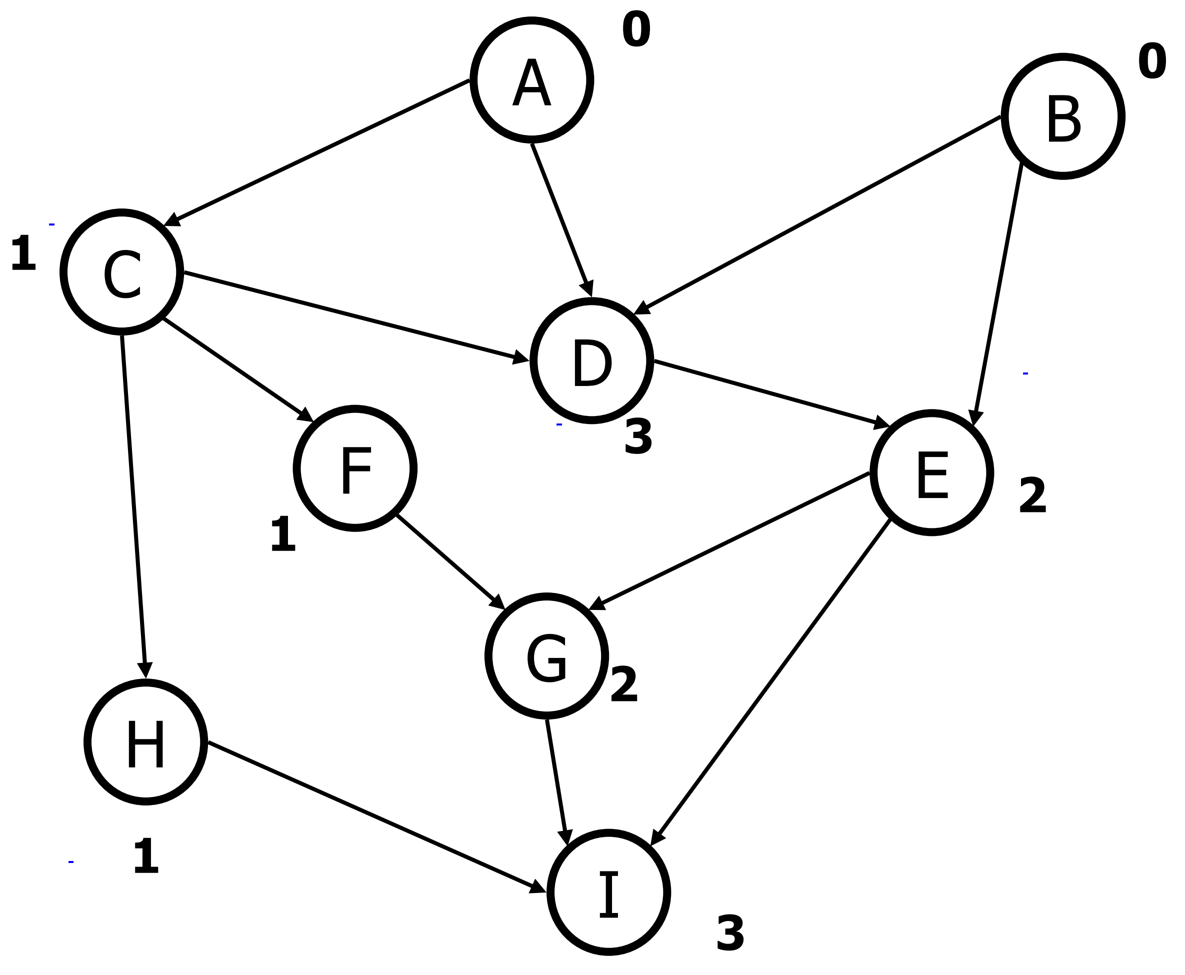
Algorithm Directed BFS

```
Algorithm topologicalSort(G)
    Input: digraph G with n vertices
    Output: topological ordering of G or an indication of a directed cycle
    S ← Empty stack
    for each vertex u ∈ G do
        incounter(u) ← indeg(u)
        if incounter(u) = 0 then
            S.push(u)
    i ← 1
    while S is not empty do
        u ← S.pop()
        number u as vertex v_i
        i ← i + 1
        for each outgoing edge  e ∈ G do
            w ← G.opposite(u,e)
            incounter(w) ← incounter(w) - 1
            if incounter(w) = 0 then
                S.push(w)
    if i > n then
        return v_1, v_2, ..., v_n
    return "G has a directed cycle"
```

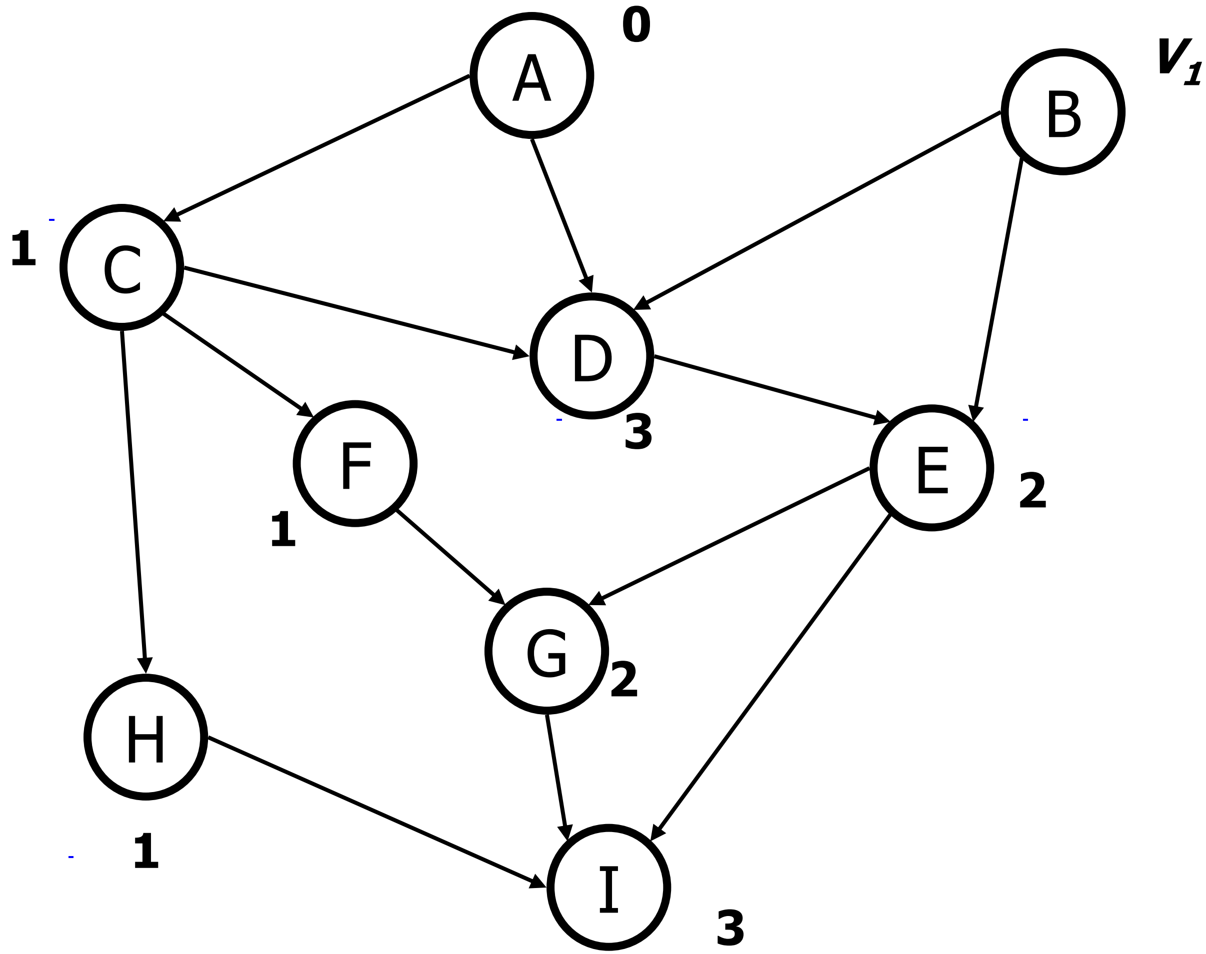
Topological Sorting Example



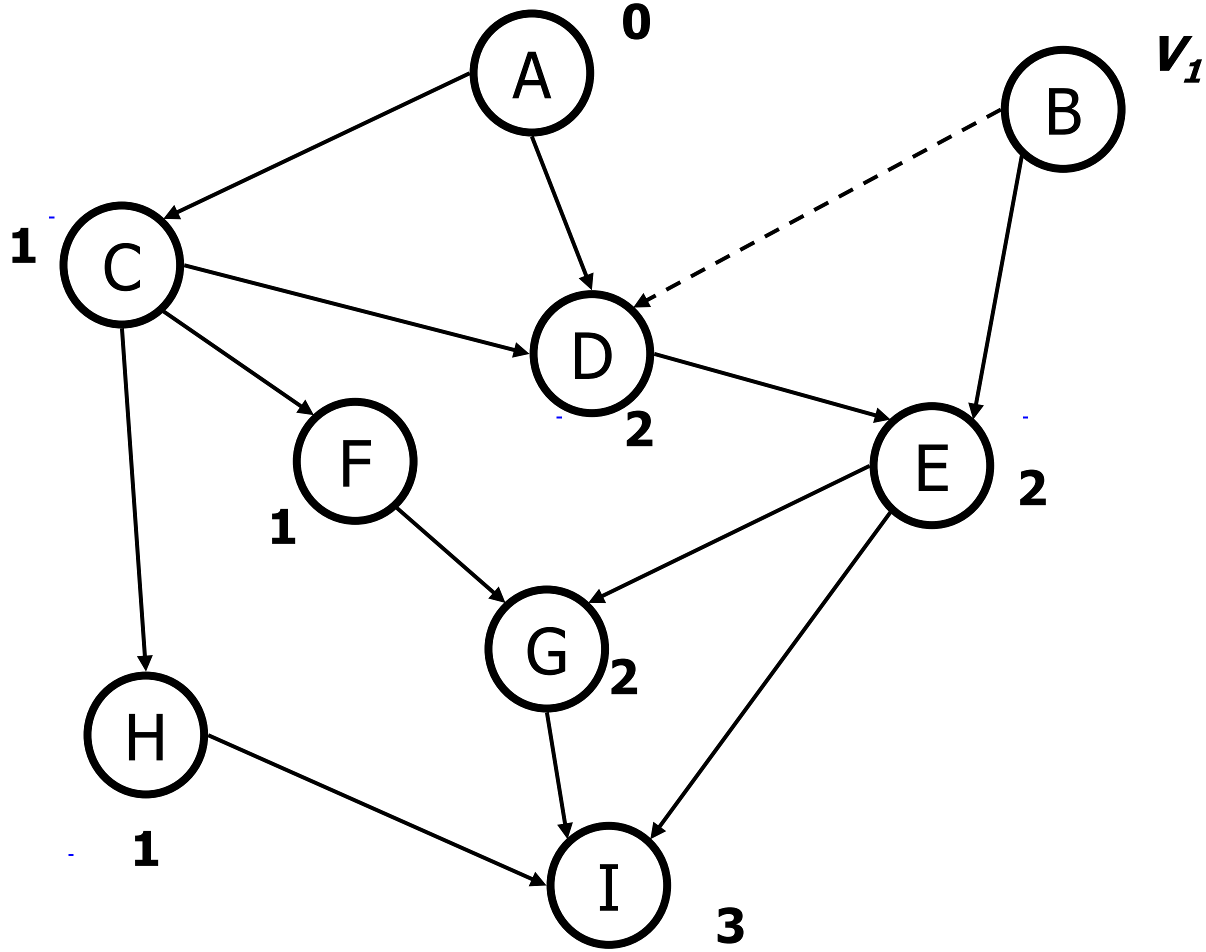
Topological Sorting Example



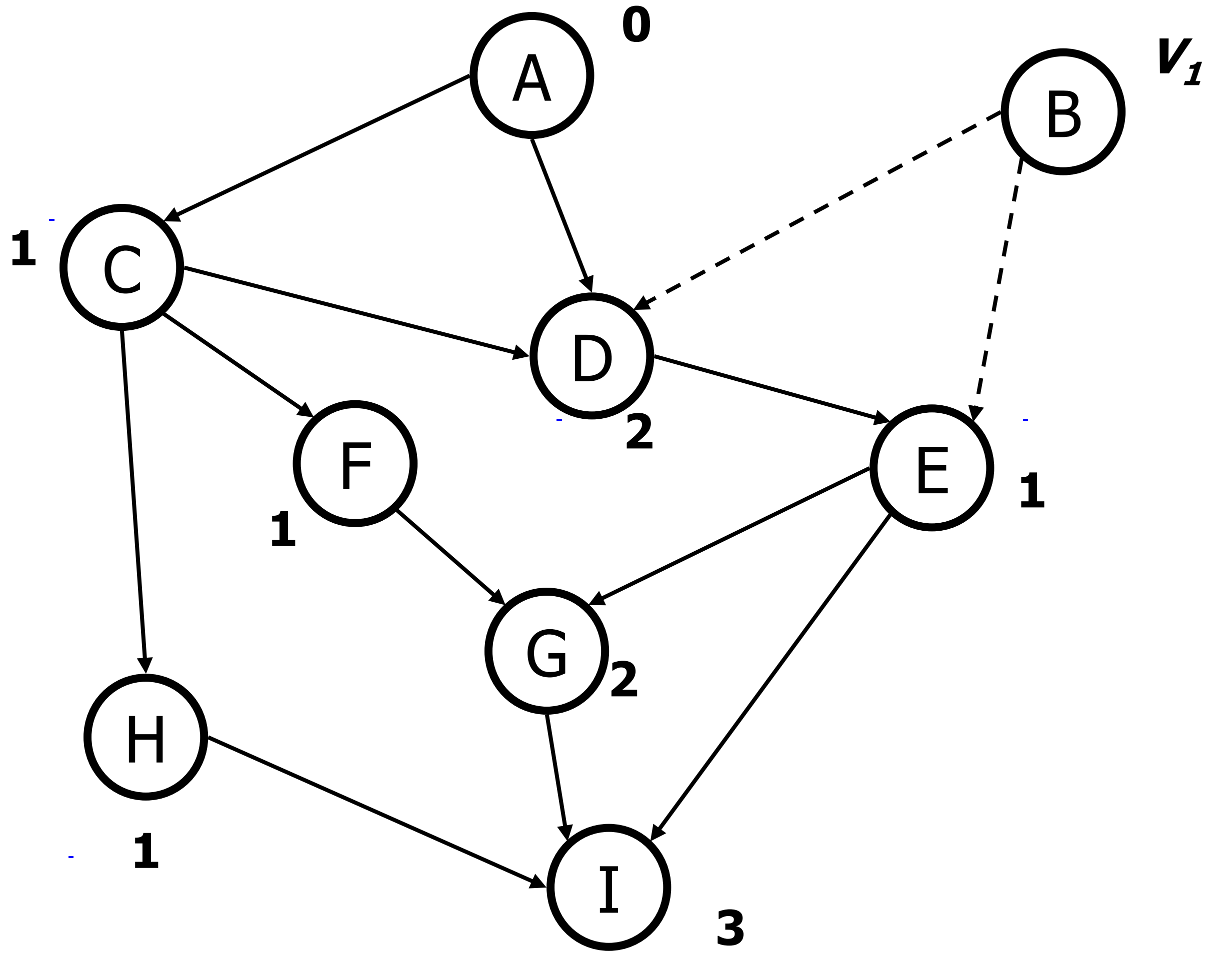
Topological Sorting Example



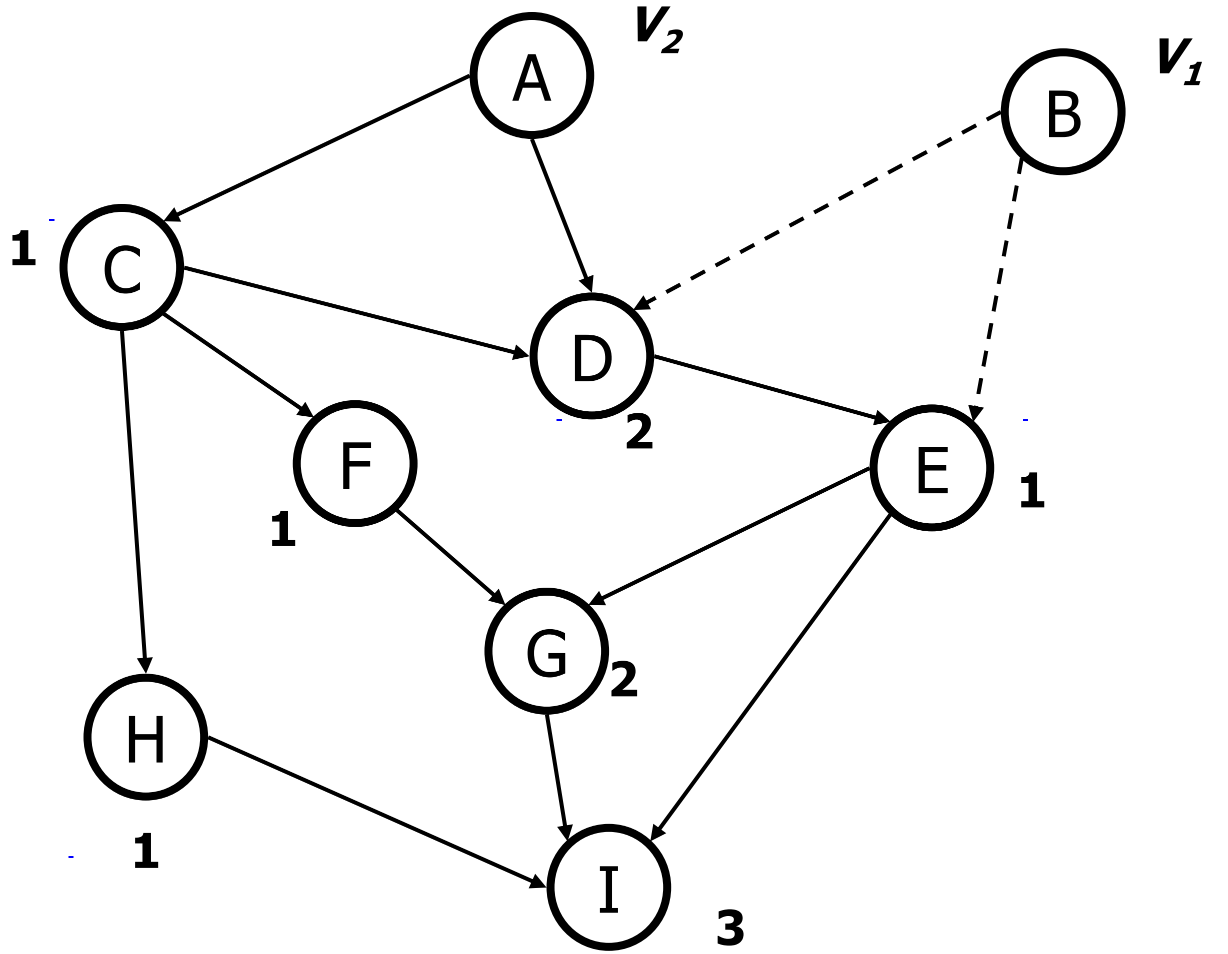
Topological Sorting Example



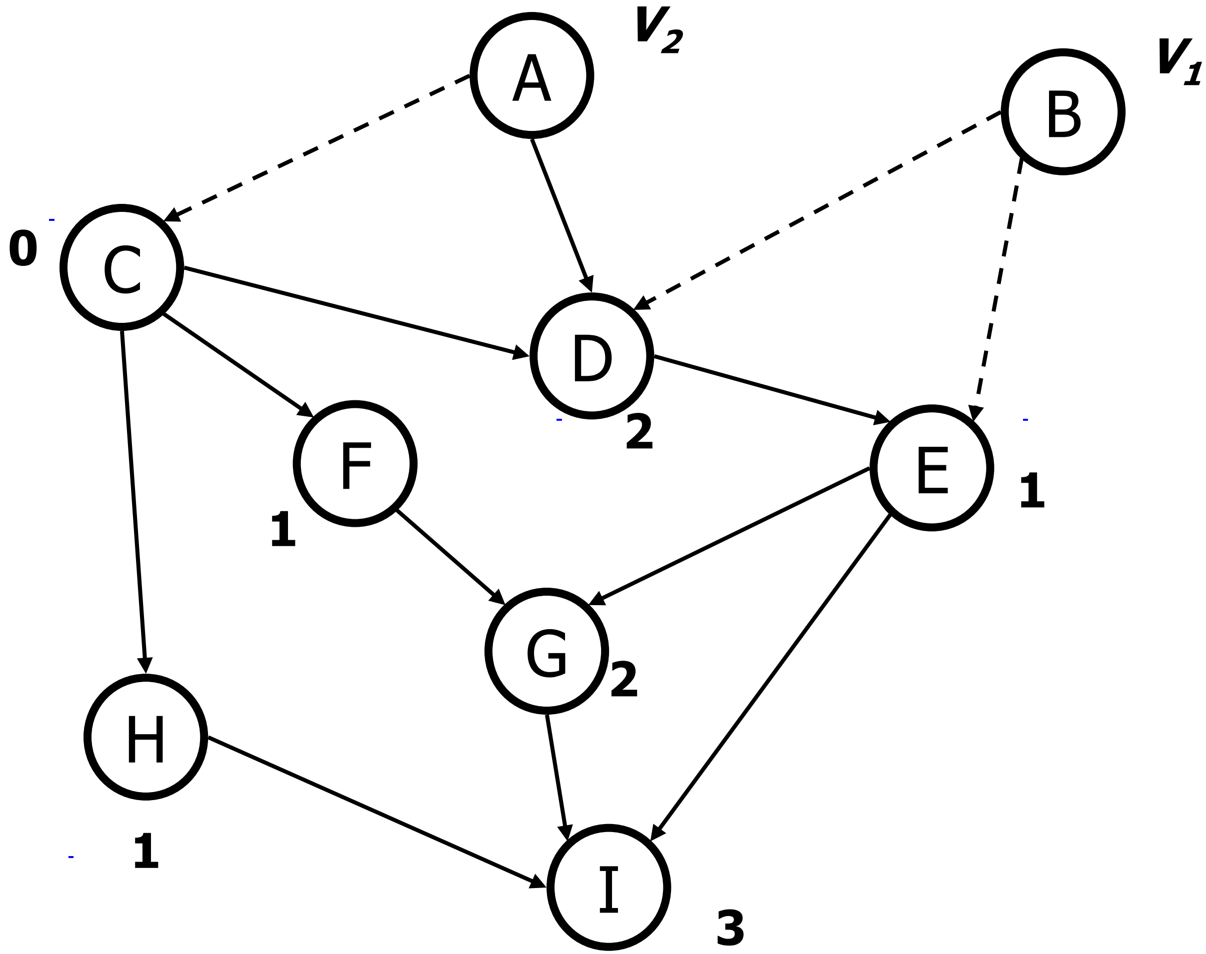
Topological Sorting Example



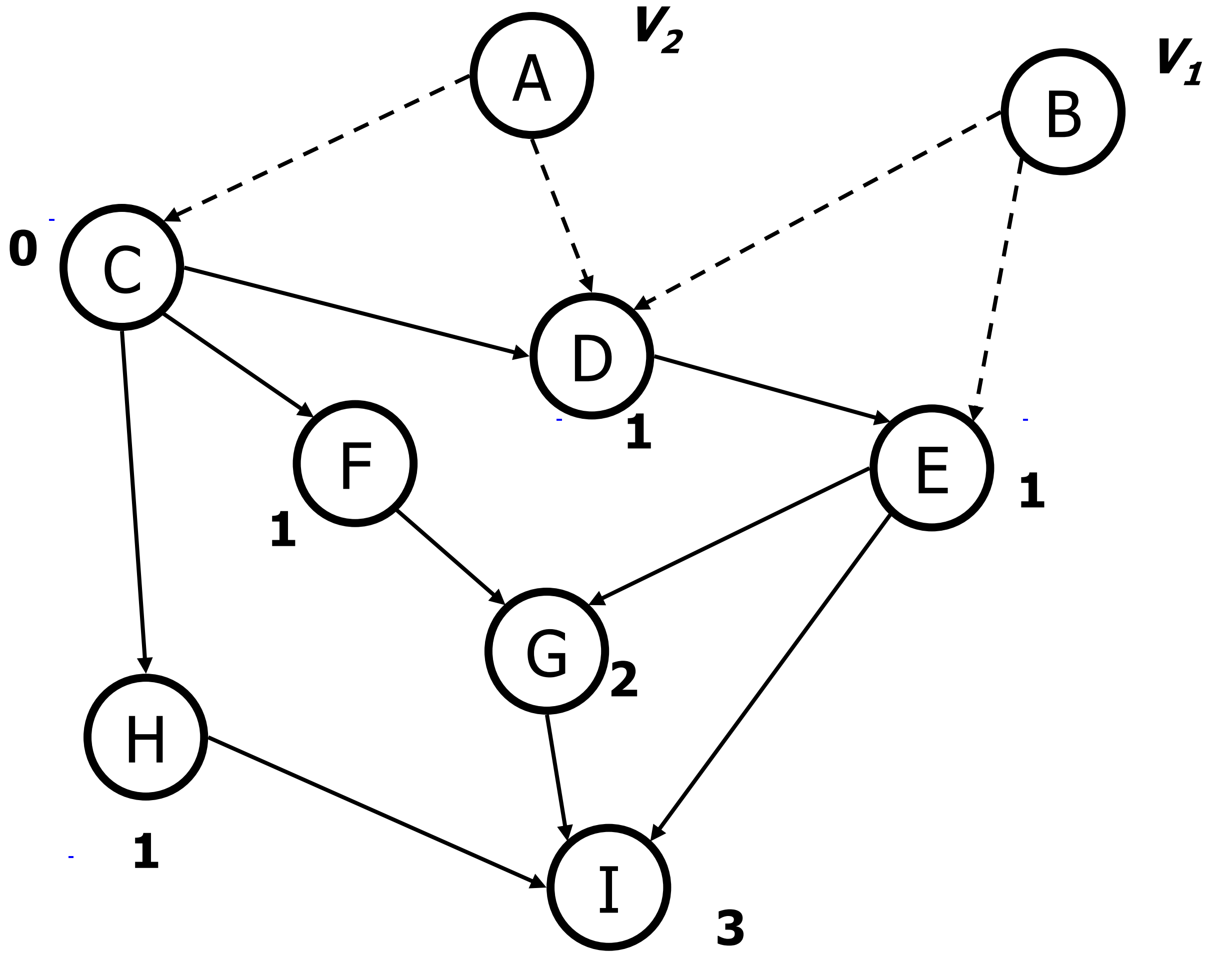
Topological Sorting Example



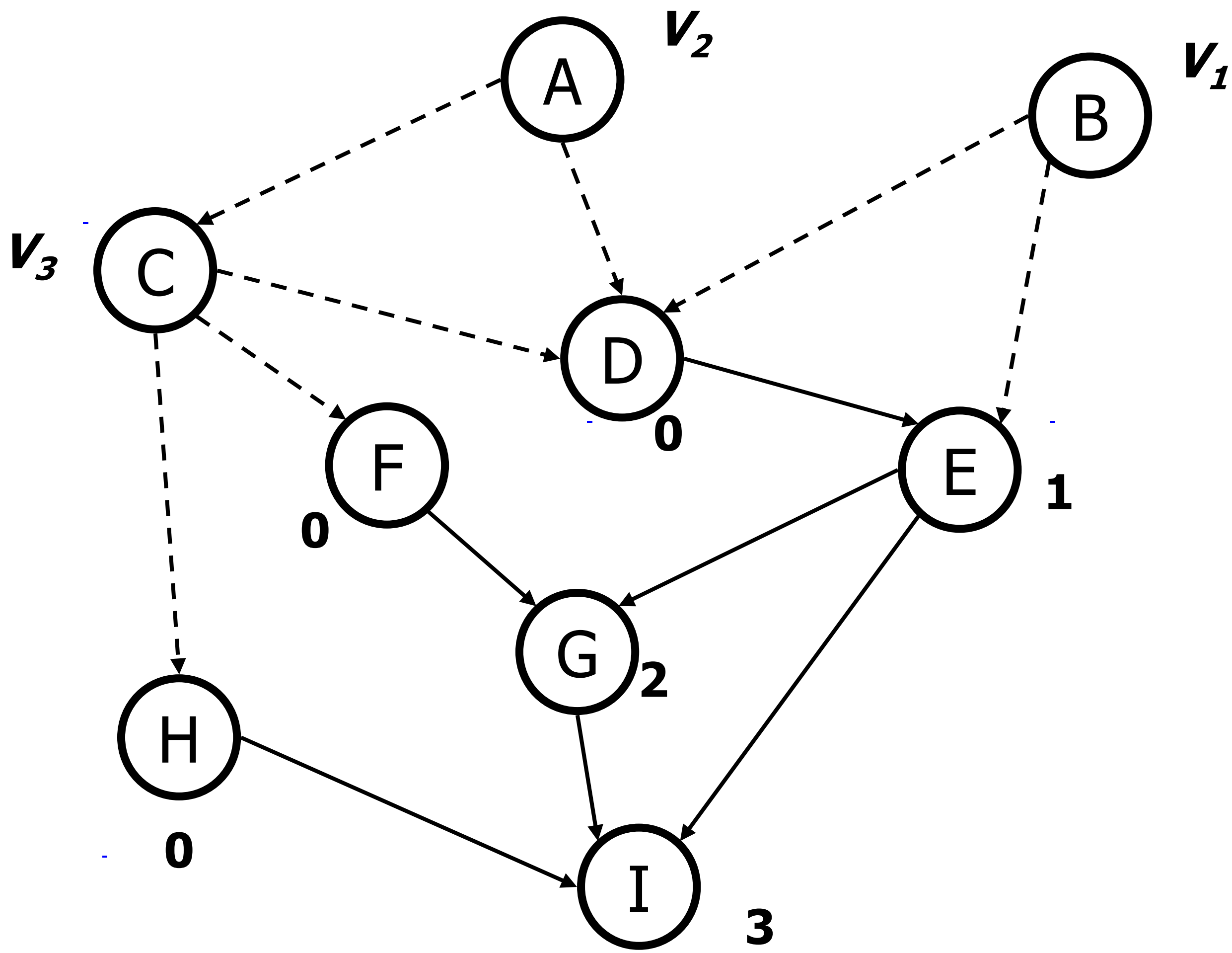
Topological Sorting Example



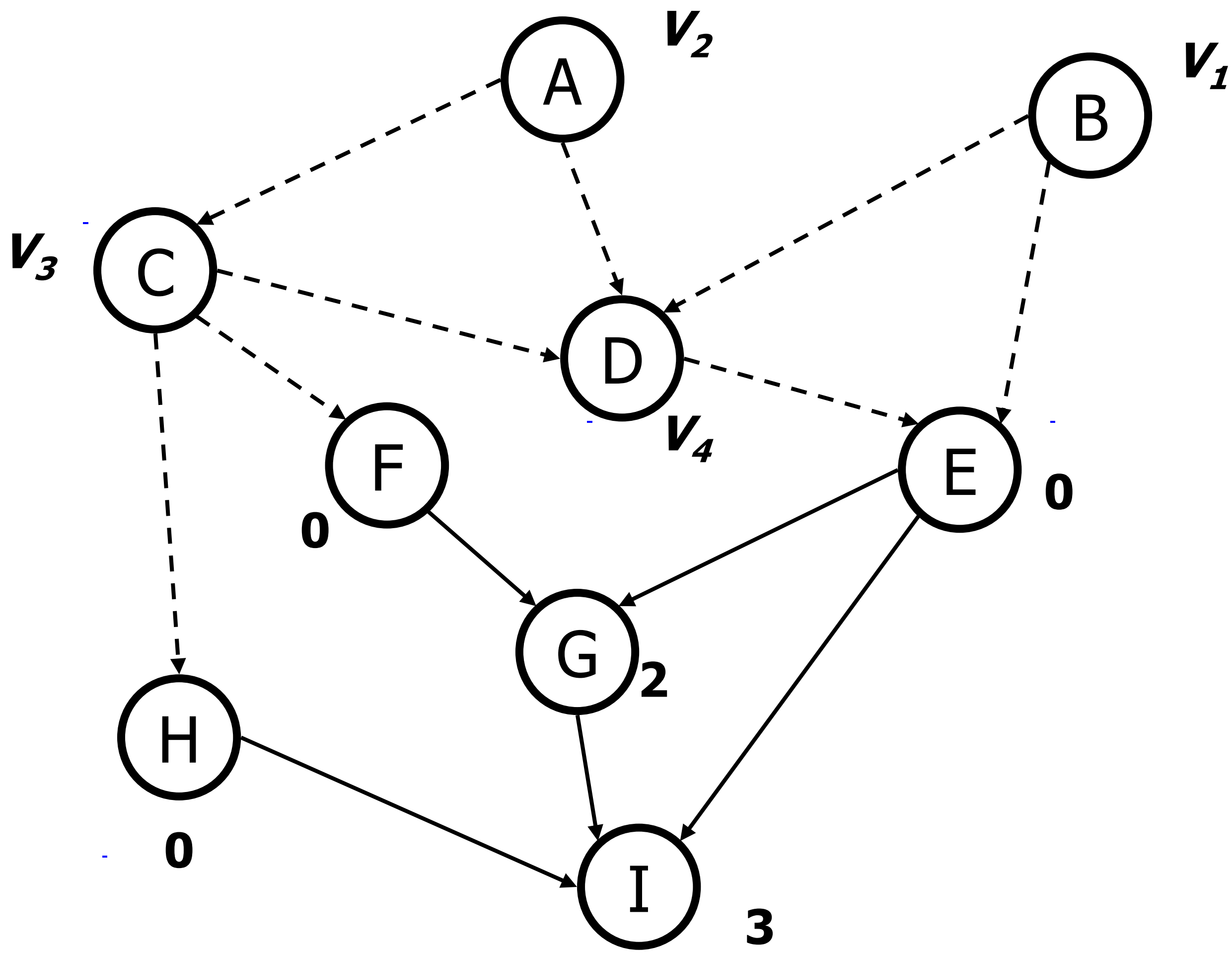
Topological Sorting Example



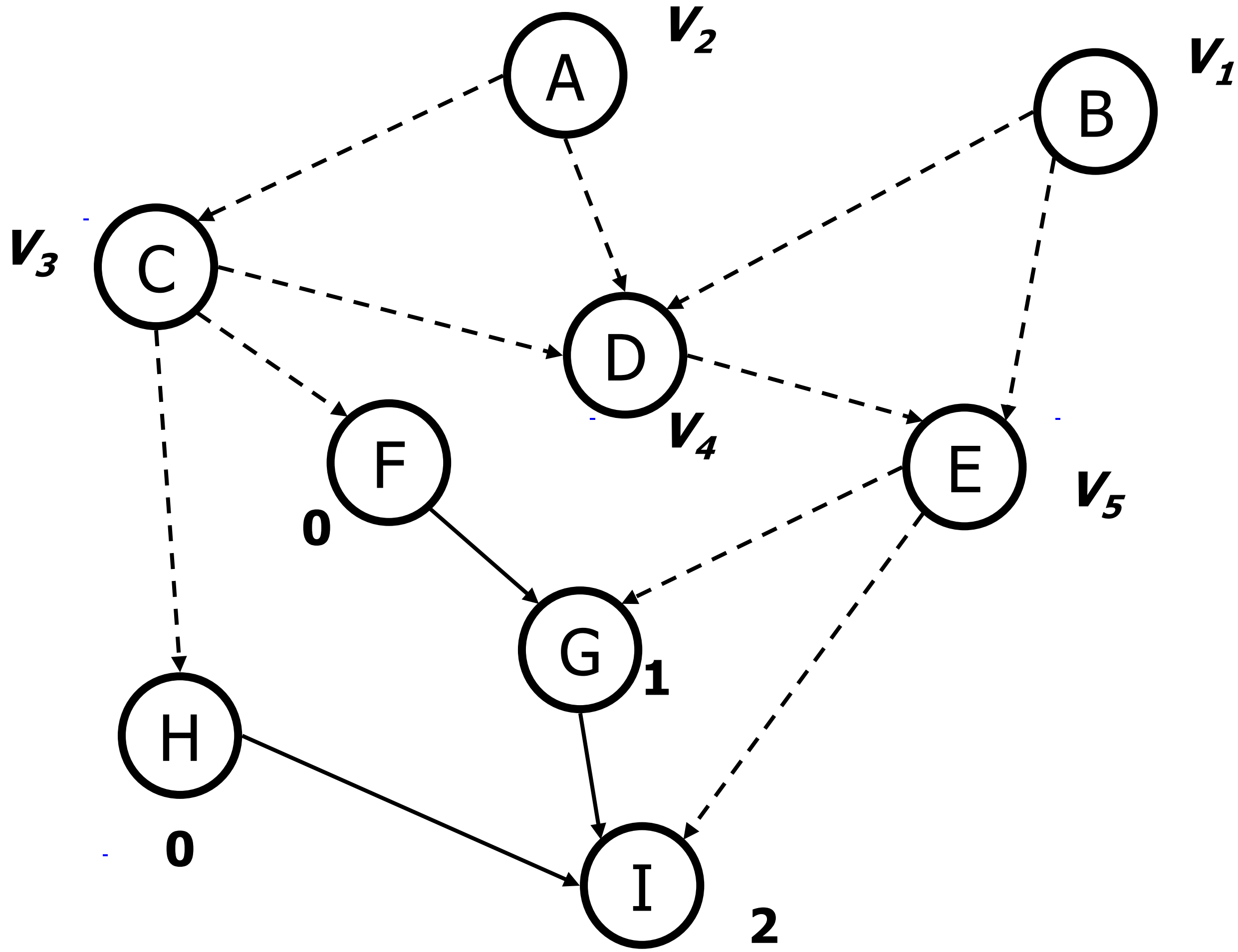
Topological Sorting Example



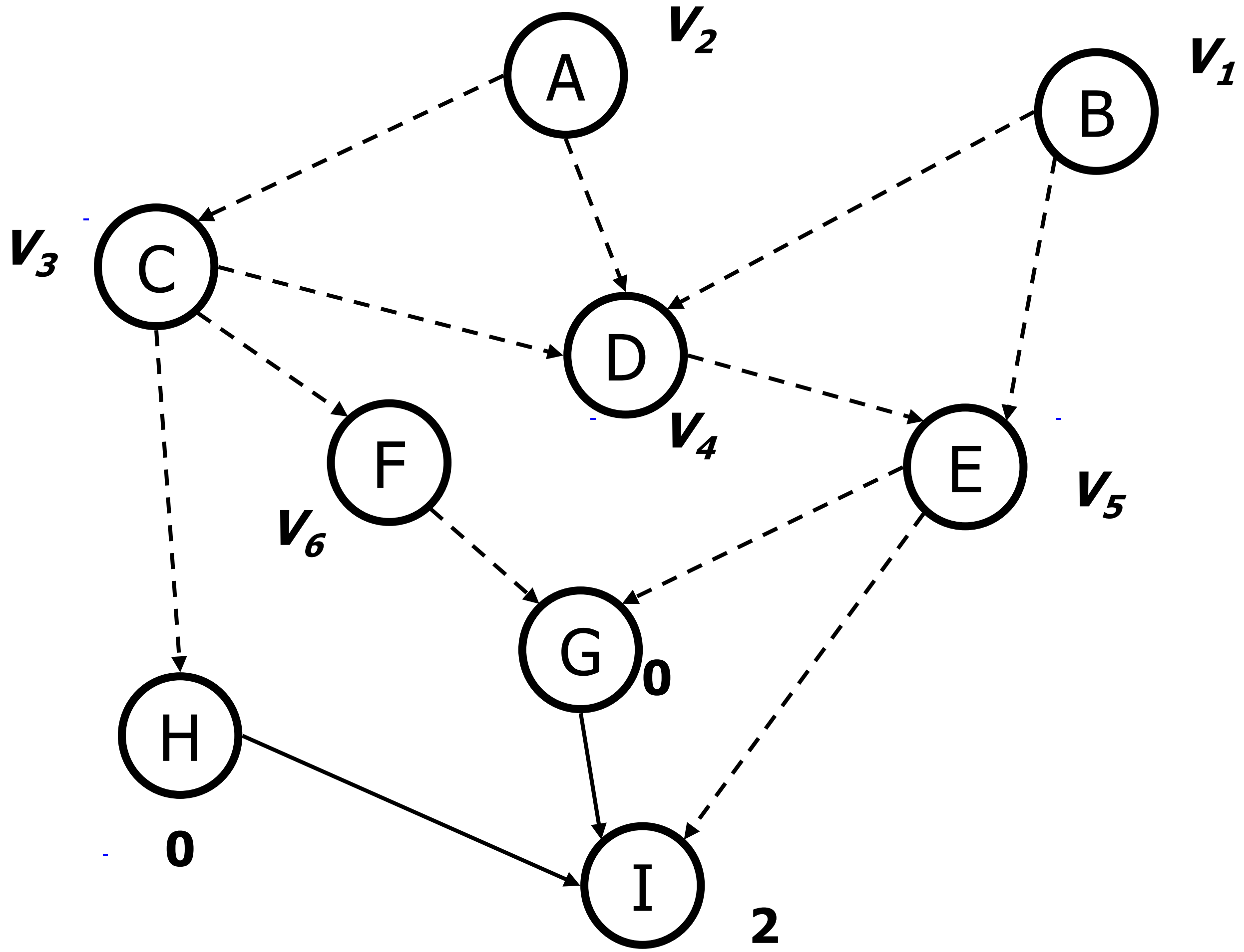
Topological Sorting Example



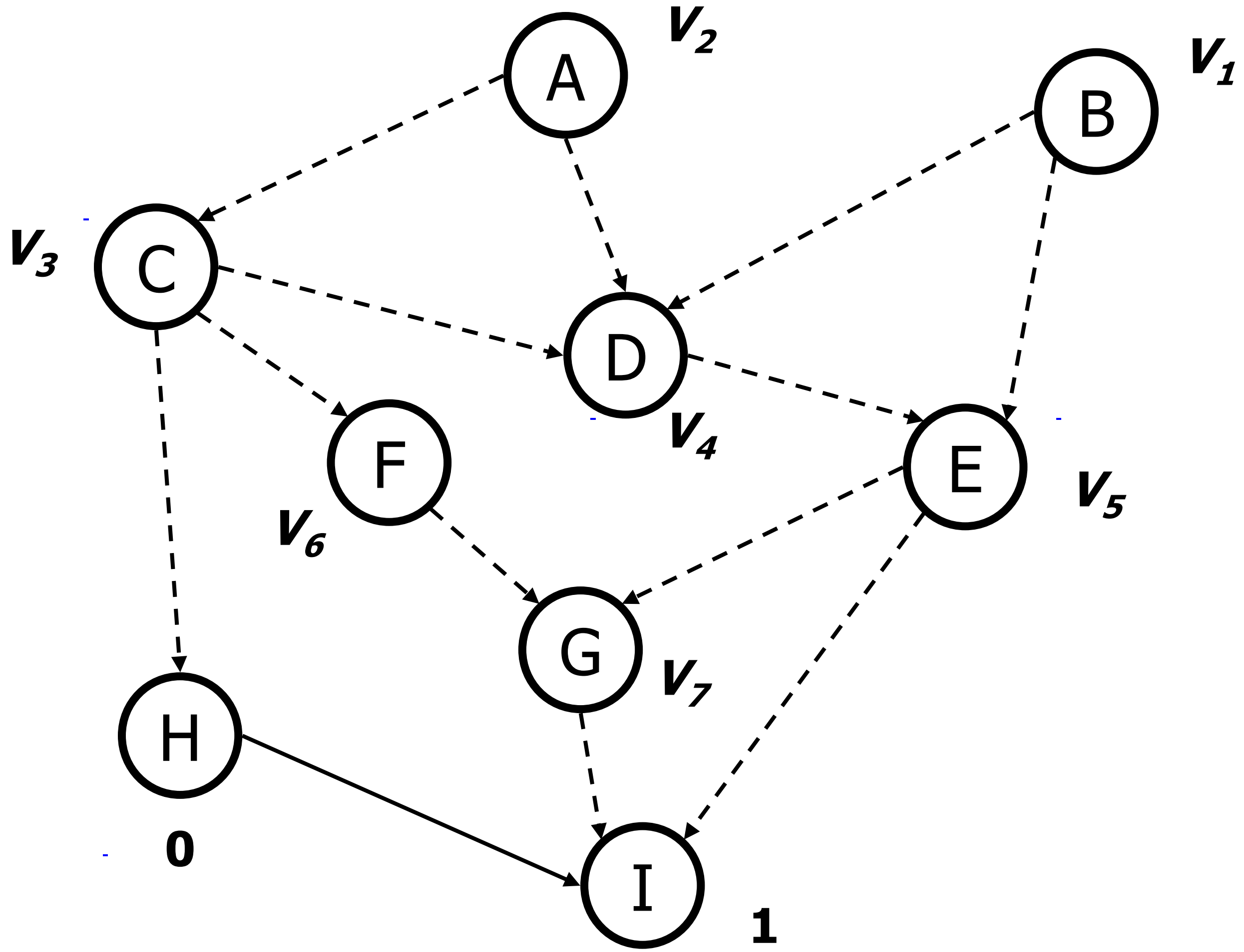
Topological Sorting Example



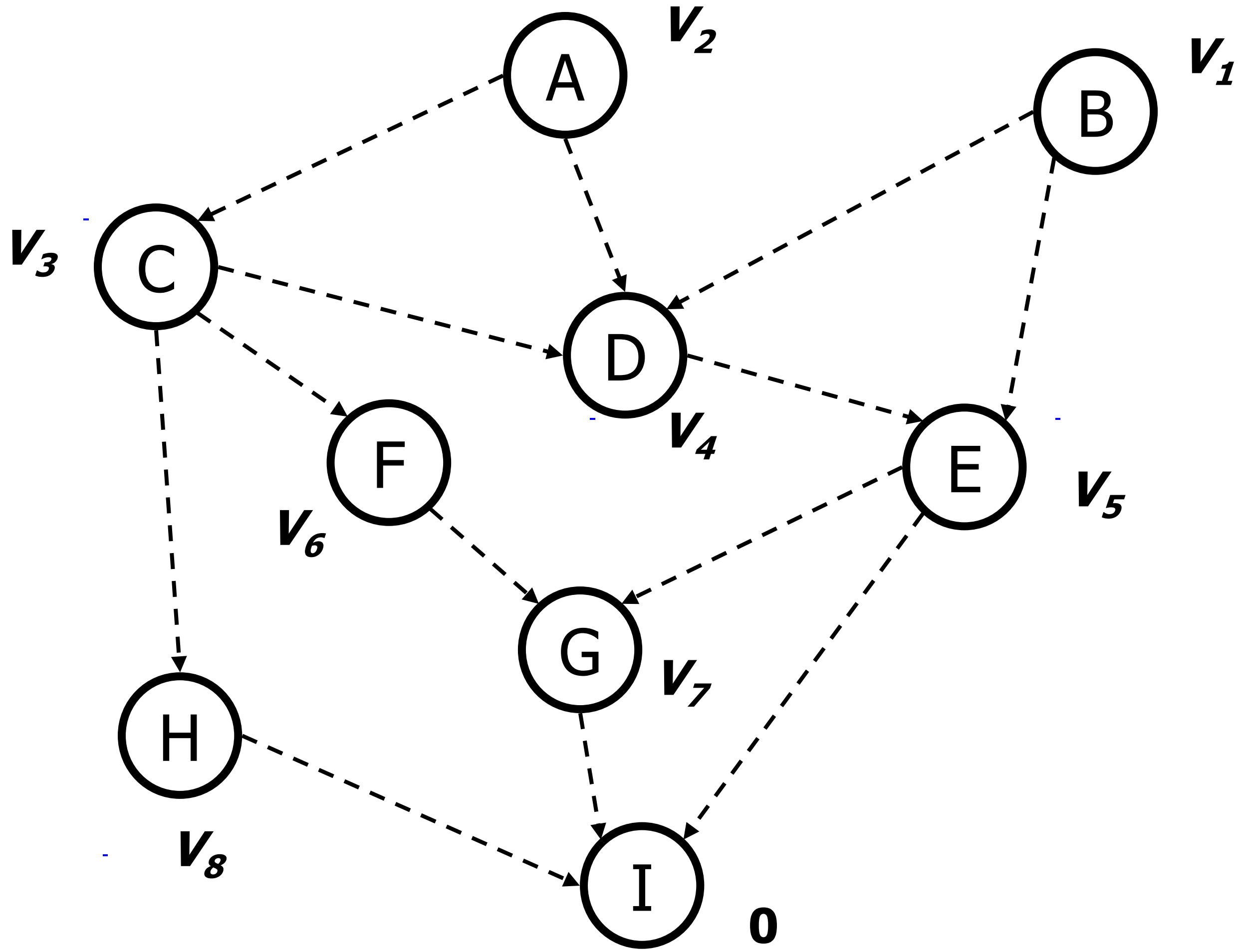
Topological Sorting Example



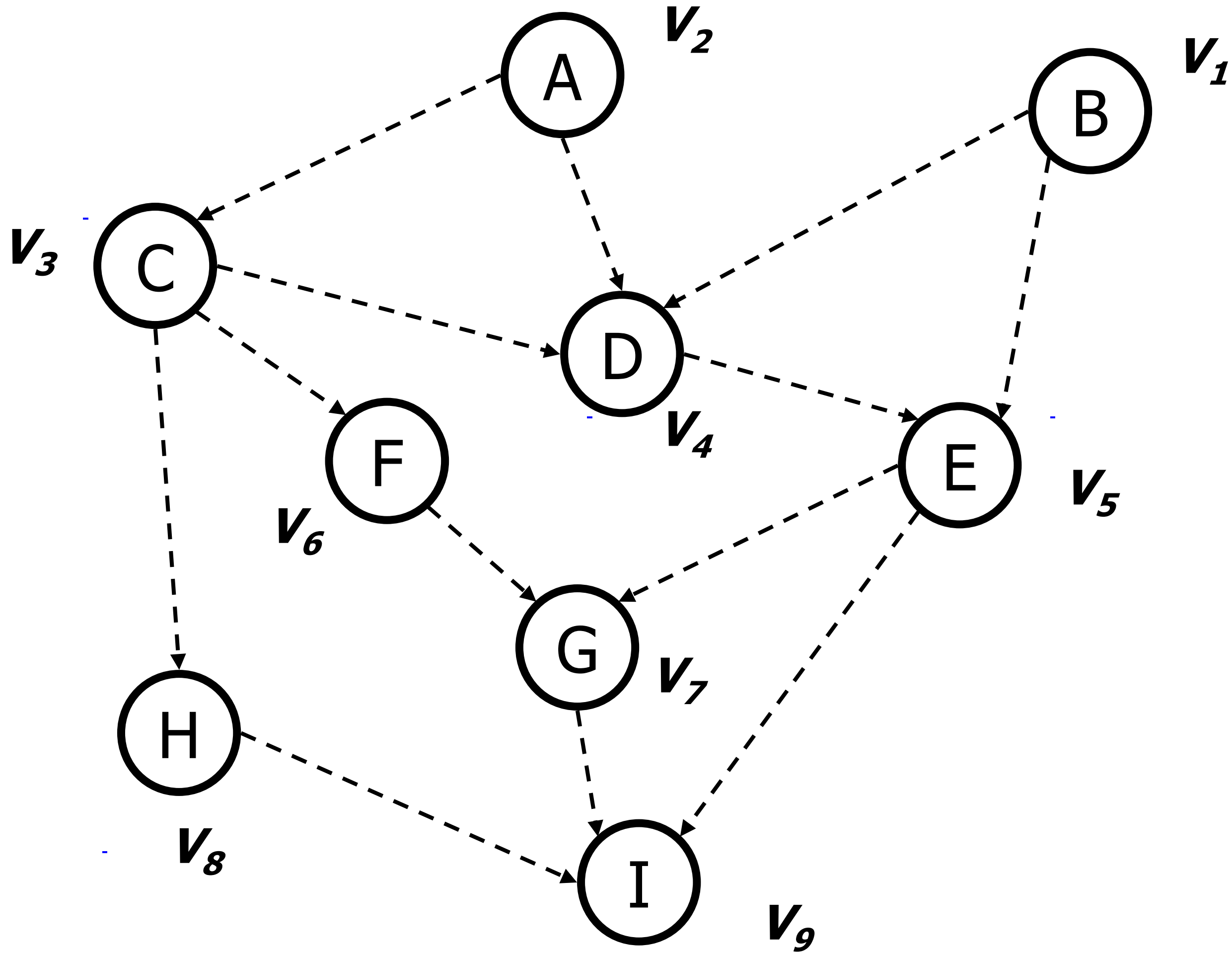
Topological Sorting Example



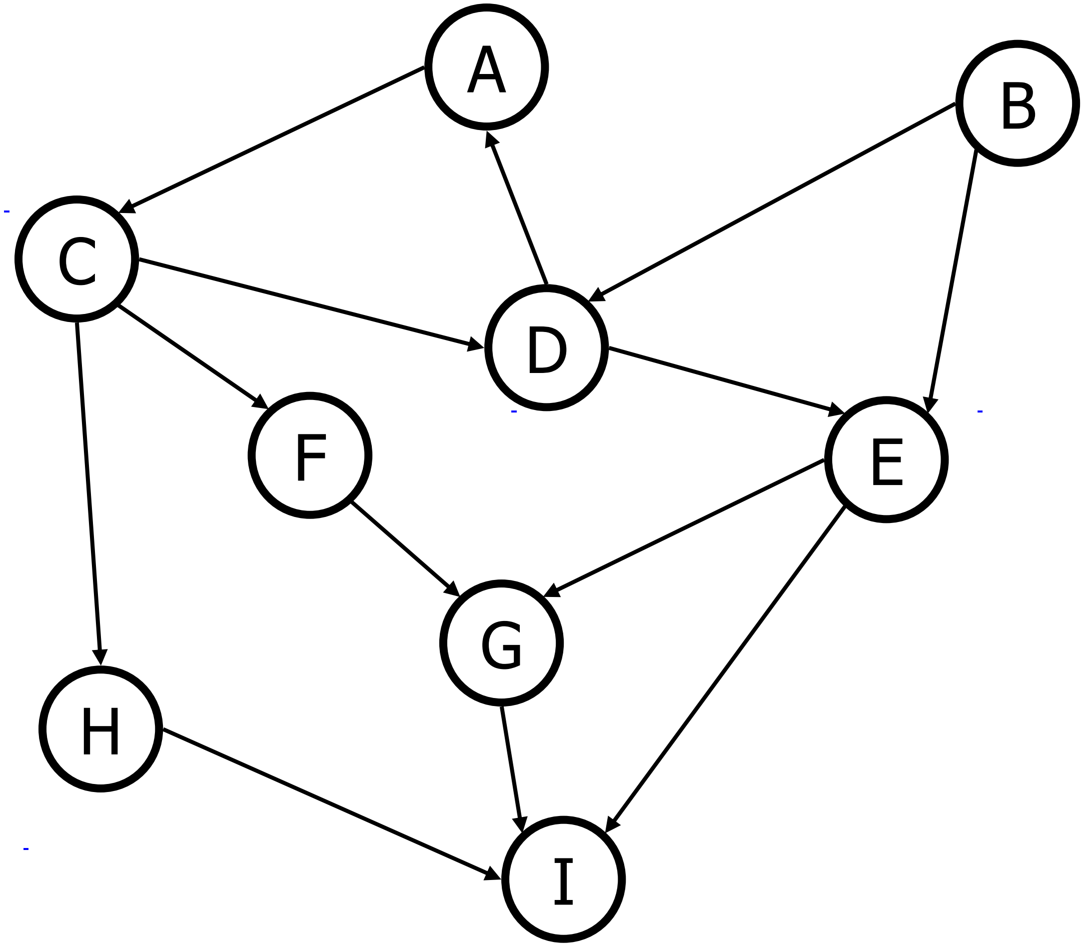
Topological Sorting Example



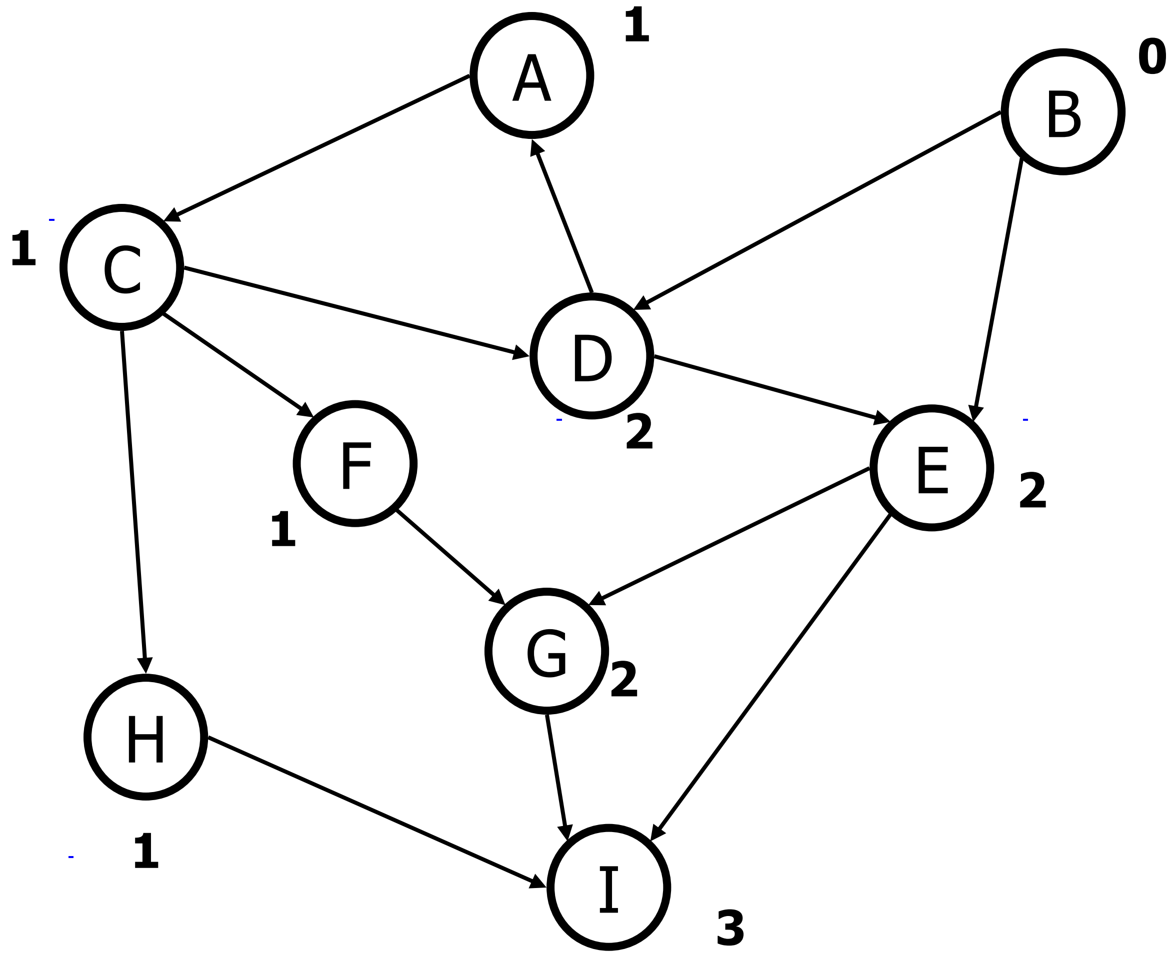
Topological Sorting Example



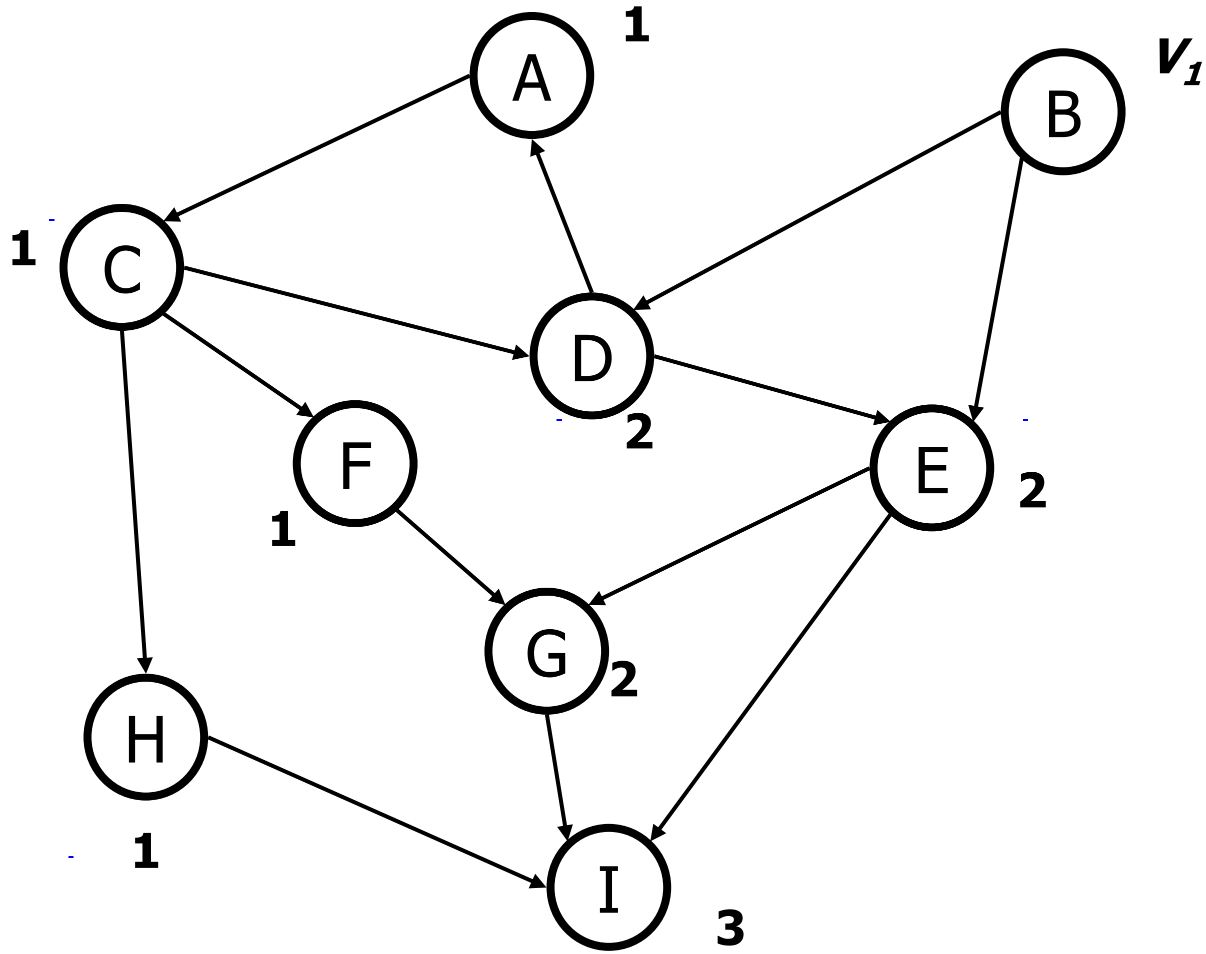
What about?



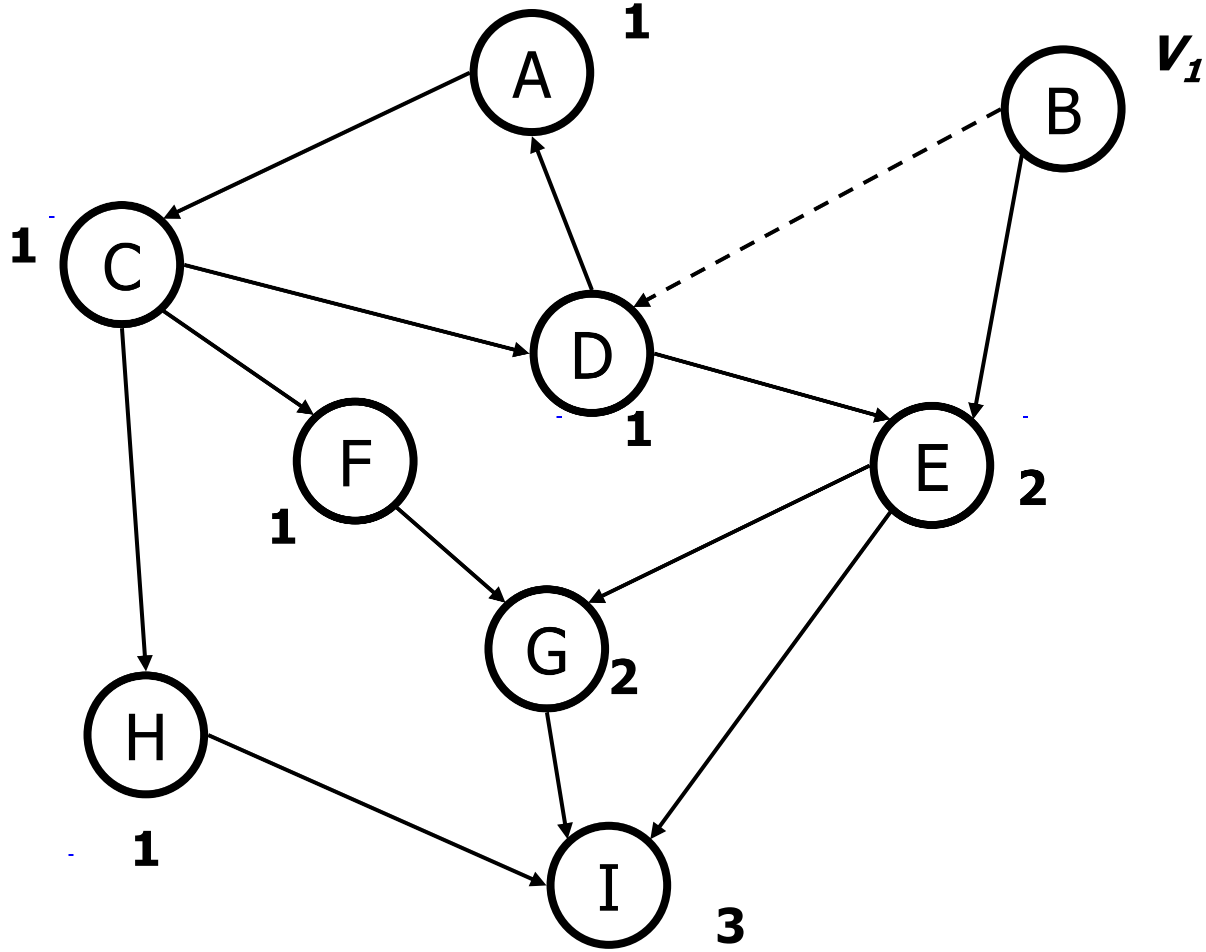
What about?



What about?



What about?



What about?