EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

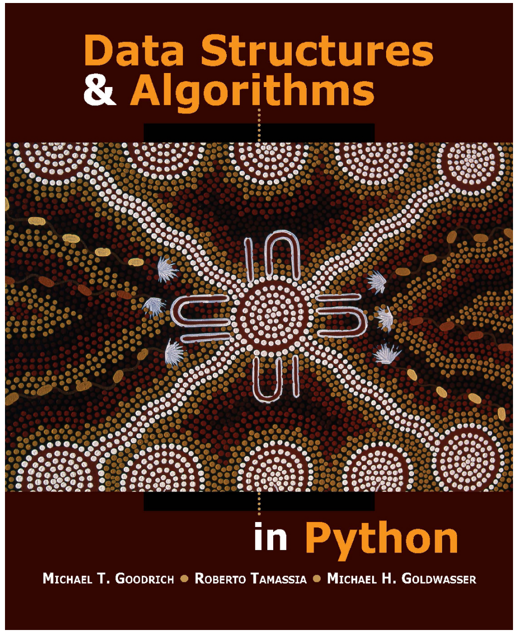
FACULTY OF
HUMANITIES
Department of General and Computational Linguistics

Directed Graphs

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Michael Goodrich
Roberto Tamassia
Michael Goldwasser



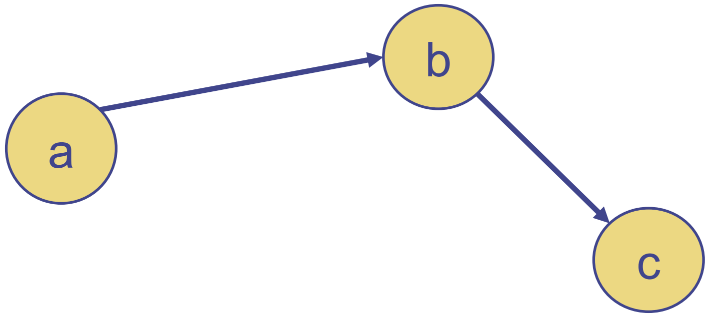
14.1-3 Digraph Properties and Traversals

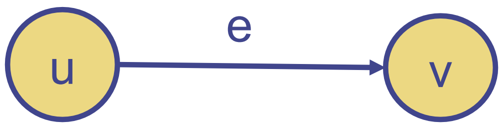
14.4 Transitive Closure

14.5 Topological Sorting

Directed Graphs

- A directed graph or digraph G is a set V of vertices – together with a collection E of pairwise connections between vertices from V, called edges where all the edges in the graph are directed
- An edge $\mathrm{e} = (u, v)$ is directed from u to v if the pair (u, v) is ordered, with u preceding v
- The first endpoint of a directed edge is called the origin, and the second endpoint is called the destination of the edge
  - u is the origin, v is the destination of edge e

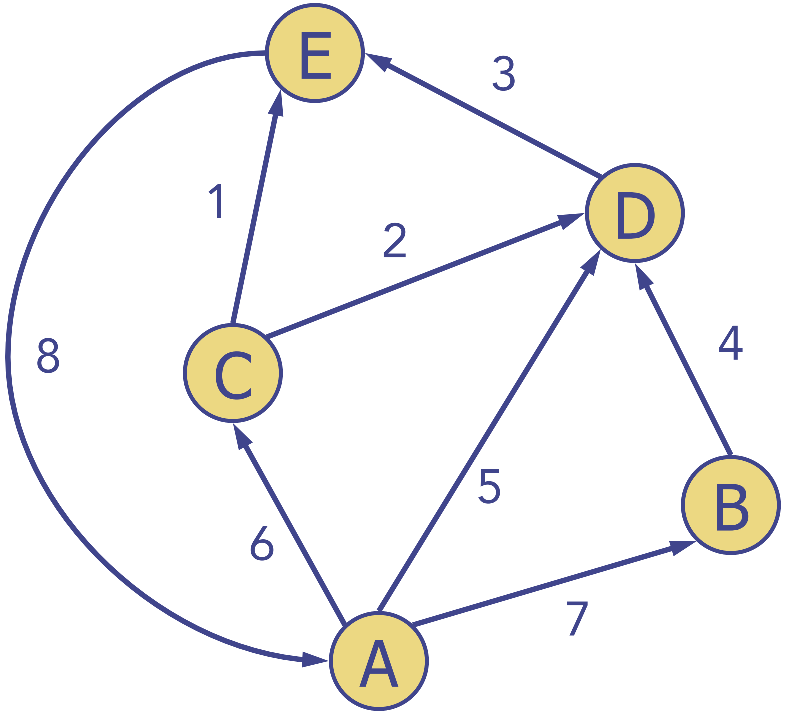




EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Directed Graphs - Terminology

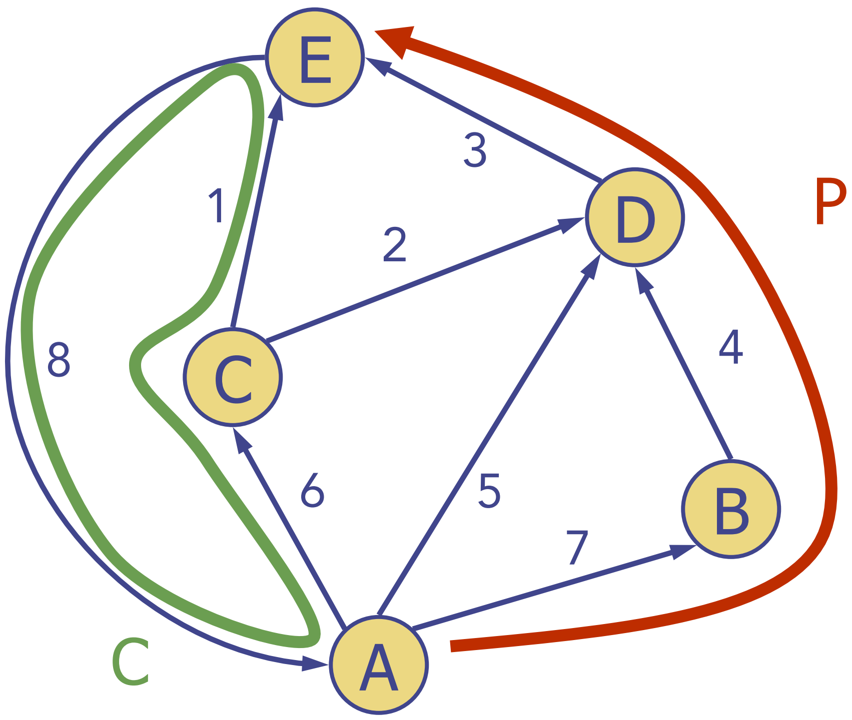
- the outgoing edges of a vertex are the edges whose origin is that vertex
  - The outgoing edges of vertex C are 1 and 2
- The incoming edges of a vertex are the edges whose destination is that vertex
  - The incoming edge of vertex C is 6
- [ug] Two vertices u and v are adjacent if there is an edge whose end vertices are u and v
- [ug] An edge is called incident to a vertex if the vertex is one of the edge's endpoints
- [ug] The degree of a vertex, $\deg(v)$, is the number of incident edges of v
- The in-degree and out-degree of a vertex v are the number of incoming and outgoing edges of v, - $\mathrm{indeg}(v)$, $\mathrm{outdeg}(v)$
  - indeg(C) = 1, outdeg(C) = 2



Directed Graph – Terminology (cont'd)

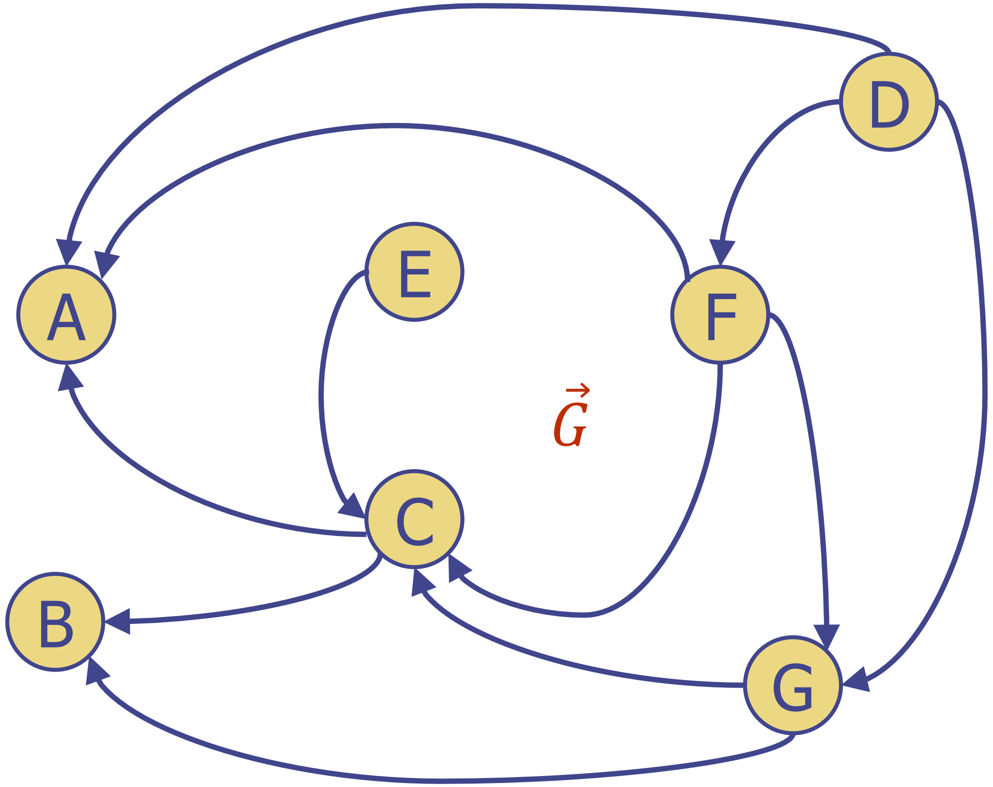
- A directed path is a path such that all edges are directed and are traversed along their direction
  - $P = (A, 7, B, 4, D, 3, E)$ is a directed simple path

- A directed cycle is a cycle such that all edges are directed and are traversed along their direction
  - $C = (E, 8, A, 6, C, 1, E)$ is a directed simple cycle

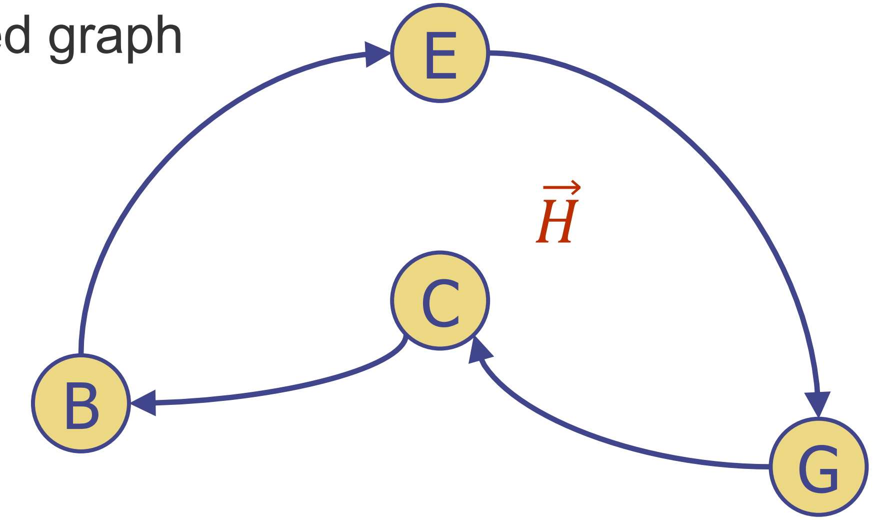


Directed Graph – Terminology (cont'd)

- A directed graph is called acyclic if it has no directed cycles
  - $\vec{G}$ is an acyclic graph



- A directed graph is strongly connected if for any two vertices u and v of $\vec{G}$, u reaches v and v reaches u
  - $\vec{H}$ is a strongly connected graph



Directed Graph Properties

- Property 1. If $\vec{G}$ is a directed graph with m edges and vertex set V, then

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

- Justification. In a directed graph each edge (u, v) contributes:
  - One unit to the out-degree of its origin u
  - One unit to the in-degree of its destination v
  - The total contribution is equal to the number of edges

Directed Graph Properties (cont'd)

- Property 2. If $\vec{G}$ is a simple directed graph with n vertices and m edges, then

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

- Justification. The graph is simple → it has no parallel edges or self-loops.
  - No two edges can have the same origin and destination
  - There are no self-loops (edges with the same origin and destination)
  - Therefore the maximum degree of a vertex is n-1
  - It follows from property 1 that $m \leq n(n-1)$

The Graph ADT

The Graph ADT – for directed graphs

- A graph is a collection of vertices and edges
- Can be modelled as a combination of three data types: `Vertex`, `Edge` and `Graph`
- class `Vertex`
  - Lightweight object storing the information provided by the user
  - The `element()` method provides a way to retrieve the stored information
- class `Edge`
  - Another lightweight object storing an associated object - the cost
  - The `element()` method provides a way to retrieve the cost of the edge
  - `endpoints()` method: returns a tuple (u, v) such that vertex u is the origin of the edge and vertex v is the destination
  - `opposite(v)` method: assuming vertex v is one endpoint of an edge (either origin or destination), return the other endpoint

The Graph ADT – for directed graphs (cont'd)

- class Graph: can be either undirected or directed – flag provided to the constuctor

`vertex_count()`	returns the number of vertices of the graph
`vertices()`	returns an iteration of all the vertices of the graph
`edge_count()`	returns the number of edges of the graph
`edges()`	returns an interation of all the edges of the graph
`get_edge(u,v)`	returns the edge from vertex u to vertex v, if one exists, otherwise None
`degree(v, out=True)`	returns the number of outgoing/incoming edges incident to vertex v, as designated by the optional parameter `out`
`incident_edges(v, out=True)`	returns outgoing edges incident to vertex v by default; report incoming edges if `out=False`
`insert_vertex(v, x=None)`	create and return a new `Vertex` storing element x
`insert_edge(u,v, x=None)`	create and return a new `Edge` from vertex u to vertex v, storing x
`remove_vertex(v)`	remove vertex v and all its incident edges from the graph
`remove_edge(e)`	remove edge e from the graph

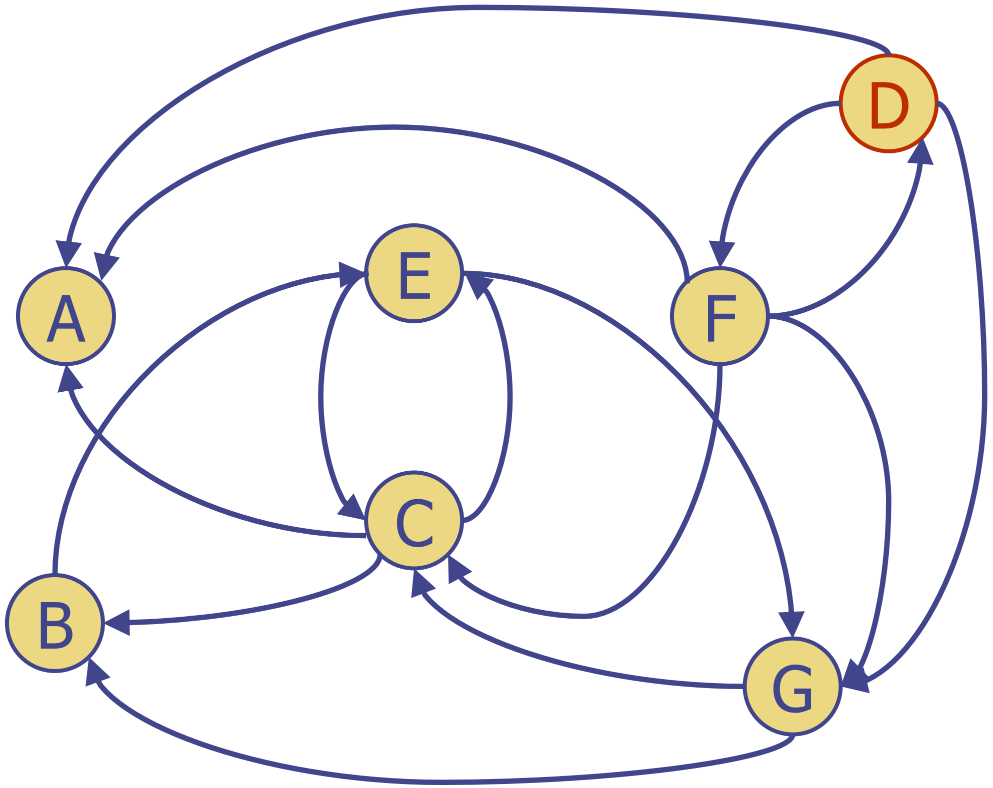
Traversals in a Directed Graph

Traversals in a Directed Graph

- The DFS and BFS techniques presented in the previous lecture for undirected graphs can be used to perform traversals of directed graphs
- The difference is that this time the edges can only be traversed from origin to destination, but not in the opposite direction
- As in the undirected graphs case, traversal algorithms can solve interesting problems dealing with reachability in a directed graph $\vec{G}$:
  - Computing a directed path from vertex u to vertex v, or report that no such path exists
  - Finding all the vertices of $\vec{G}$ that are reachable from a given vertex s
  - Determine whether $\vec{G}$ is acyclic
  - Determine whether $\vec{G}$ is strongly connected

DFS in a Directed Graph

DFS in a Directed Graph - Example

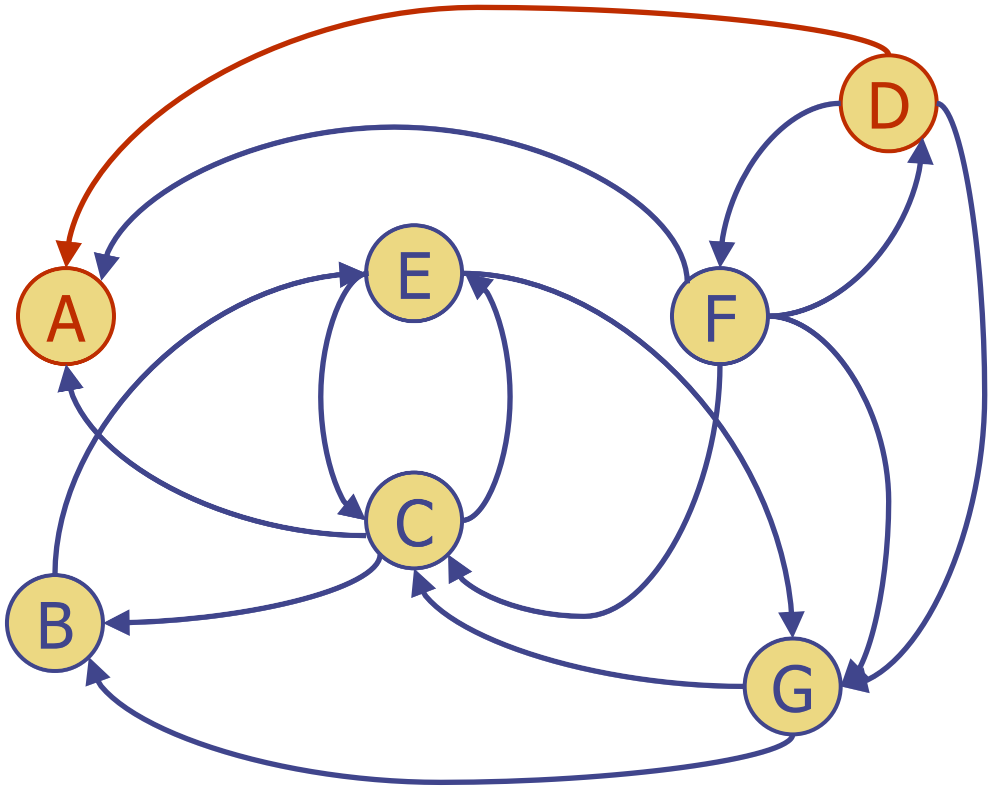


visited	discovery edge
D	None

Current vertex: D
Edges to consider: to A, F, G

- Start from vertex D, which is marked as visited (red)
- Assume that the outgoing edges of a vertex are considered in alphabetical order – e.g. for D: A, F, G

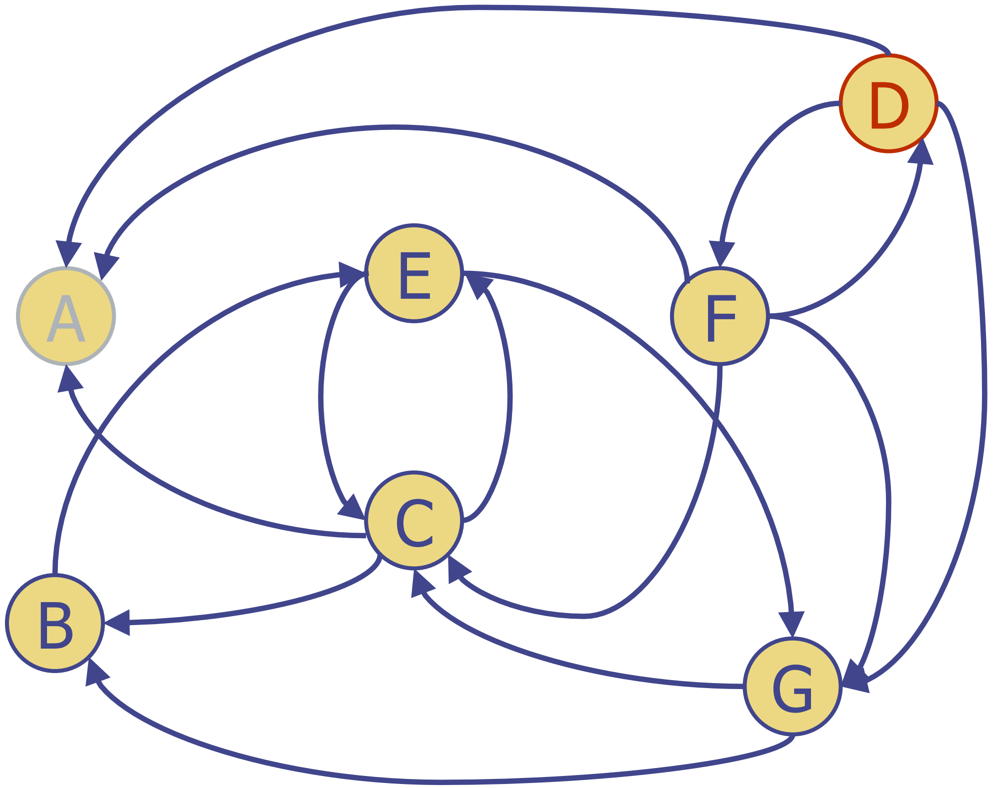
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)

Current vertex: D
Edges to consider: to F, G

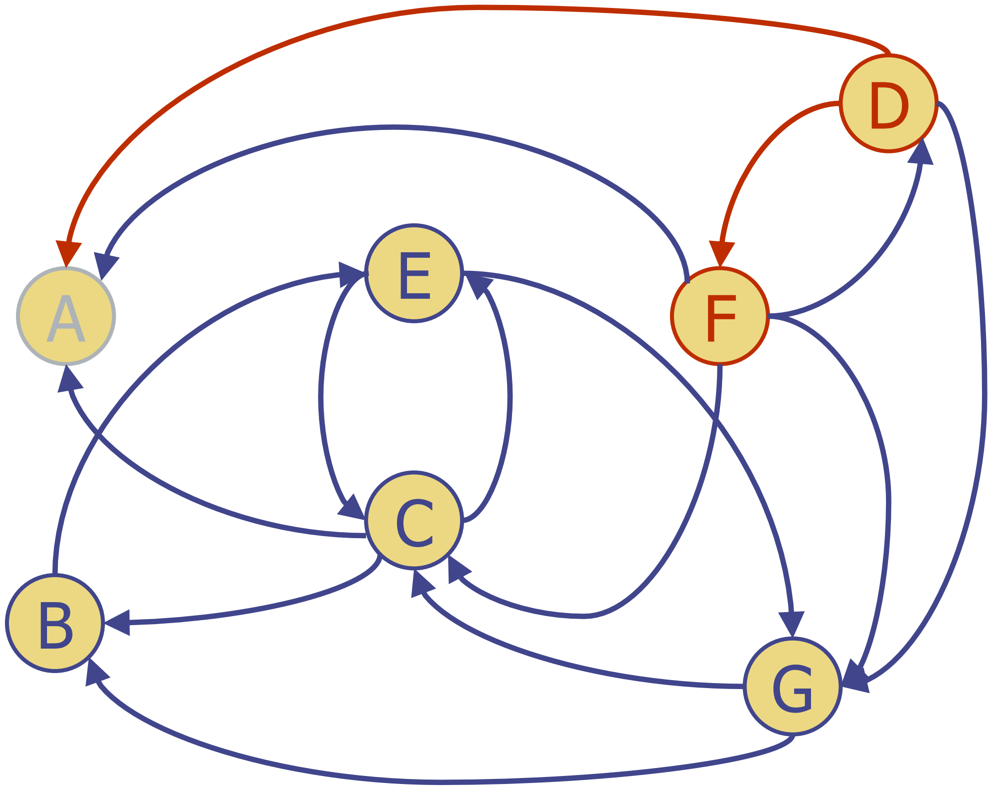
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)

Current vertex: A
Edges to consider: - (no outgoing edges)
Finished A, backtrack to D

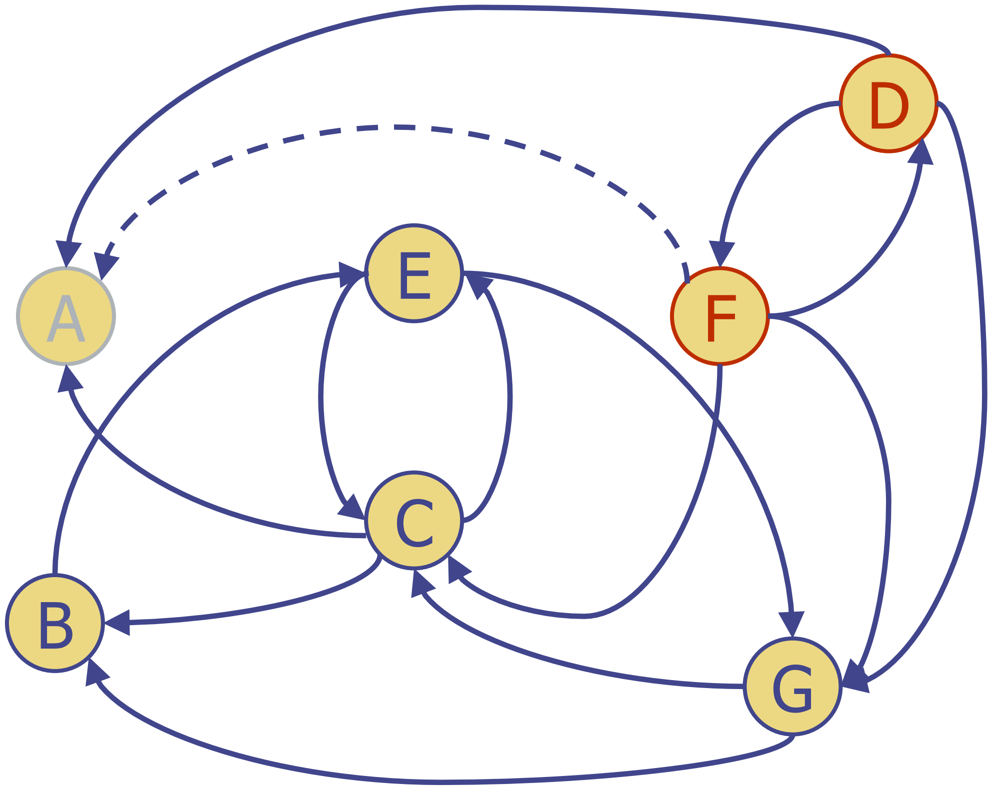
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)

Current vertex: D
Edges to consider: to F, G

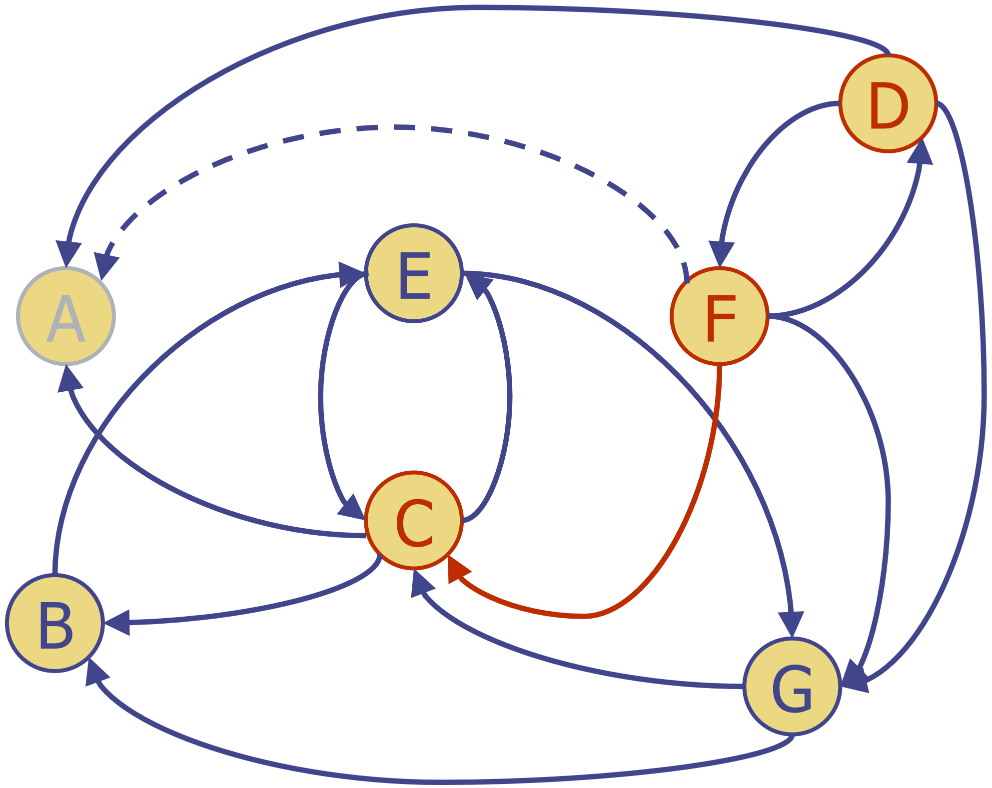
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)

Current vertex: F
Edges to consider: to A, C, D, G

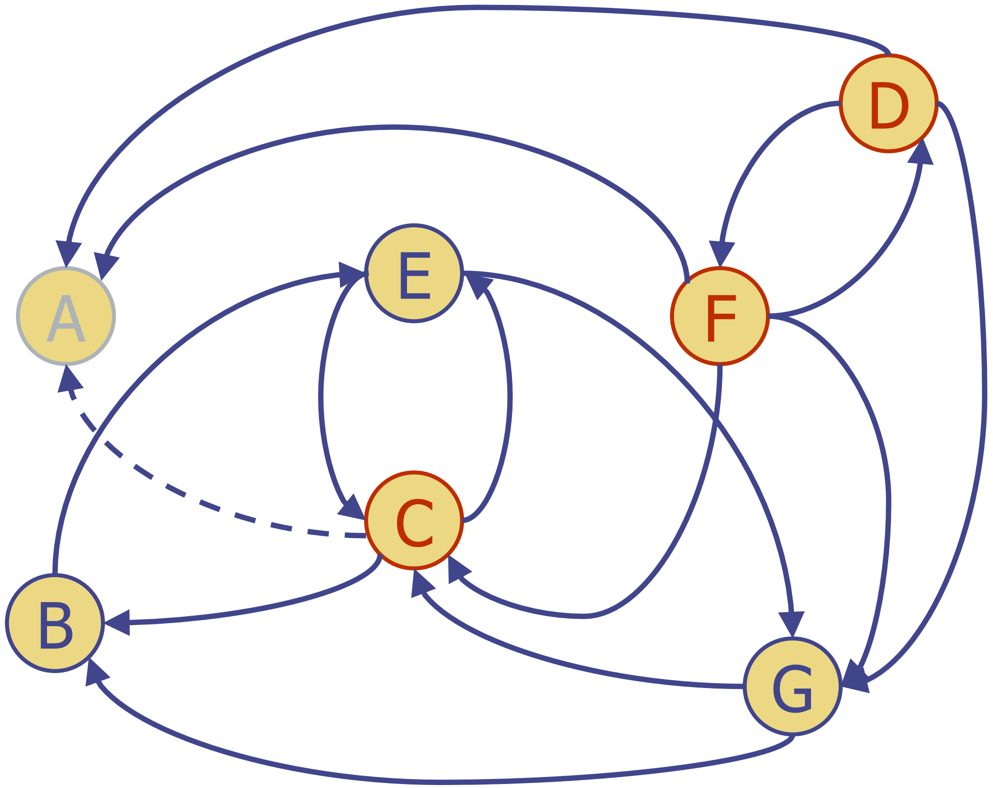
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)

Current vertex: F
Edges to consider: to C, D, G

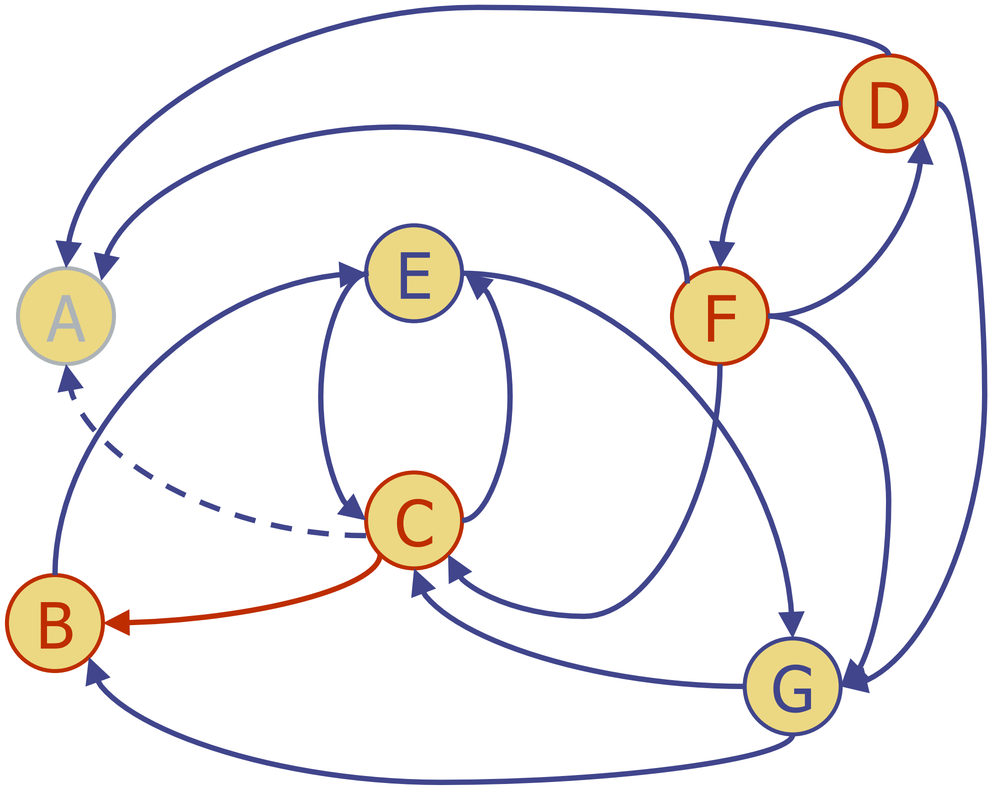
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)

Current vertex: C
Edges to consider: to A, B, E

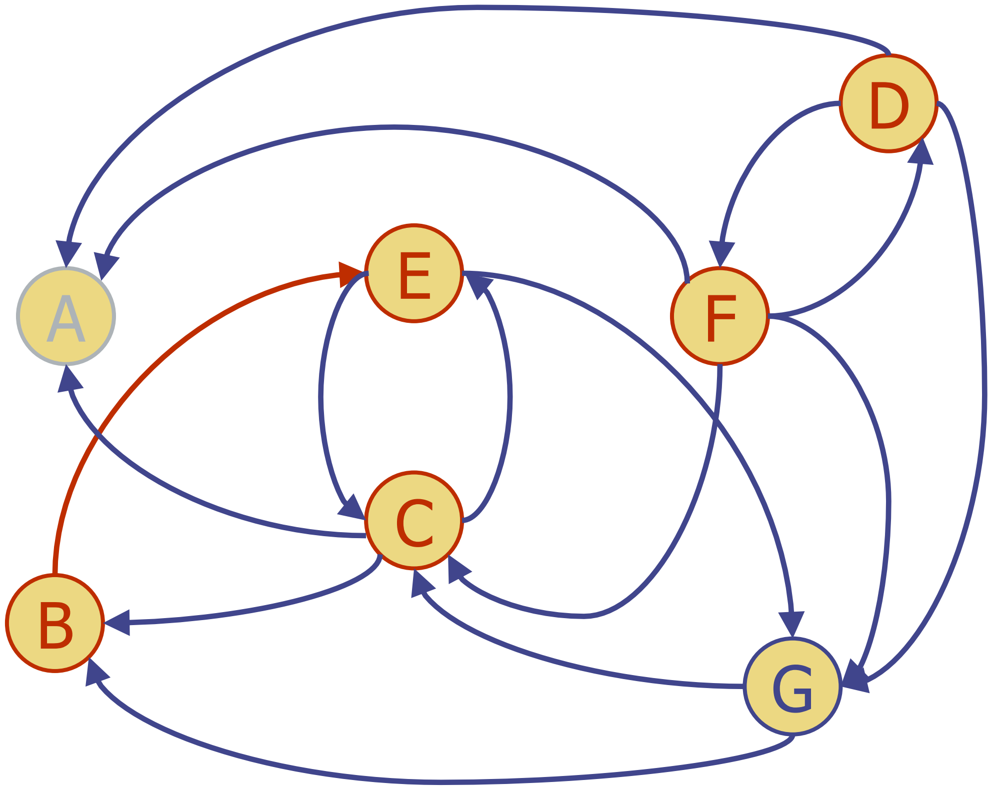
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)

Current vertex: C
Edges to consider: to B, E

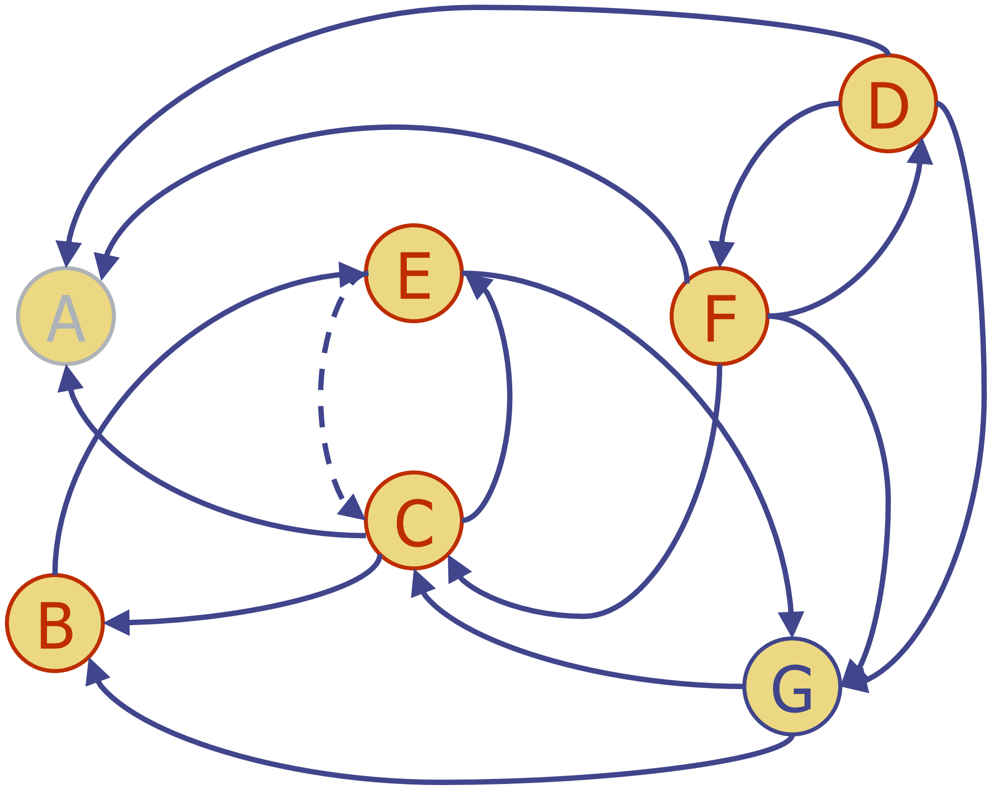
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)

Current vertex: B
Edges to consider: to E

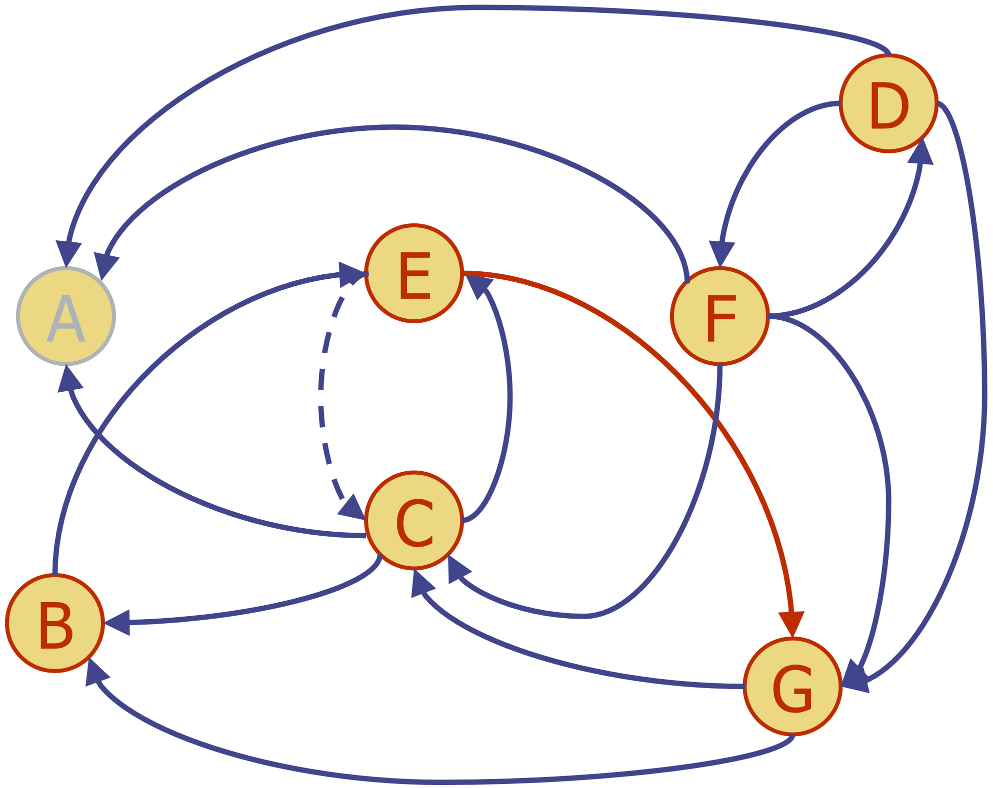
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)

Current vertex: E
Edges to consider: to C, G

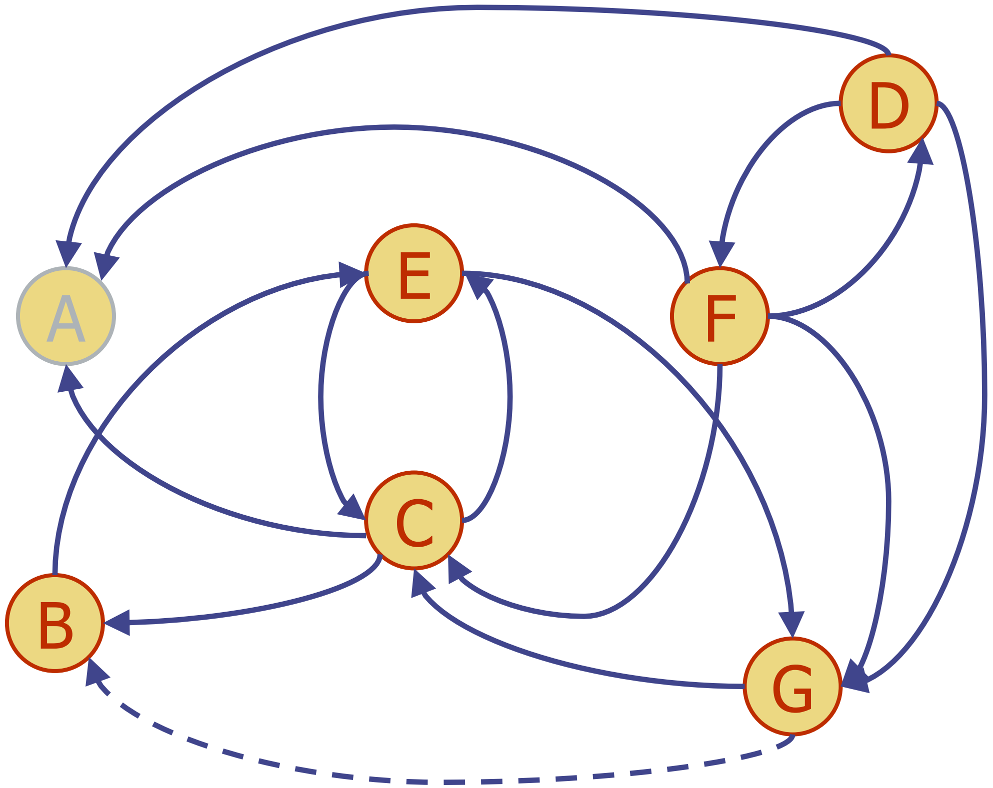
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: E
Edges to consider: to G

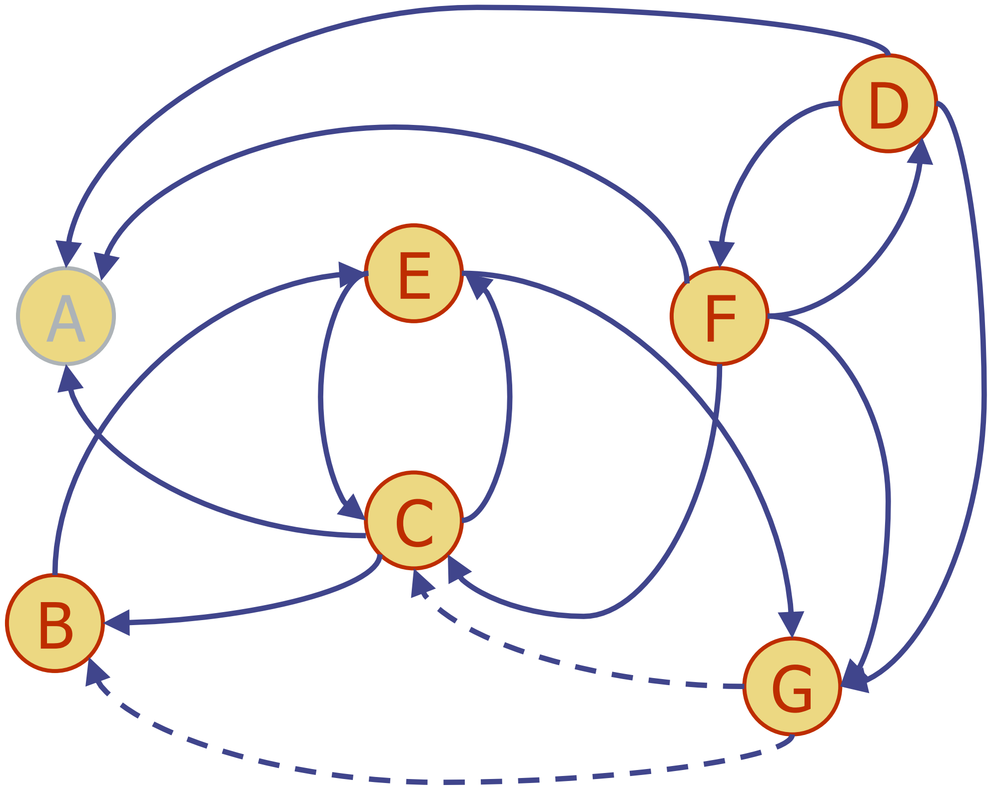
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: G
Edges to consider: to B, C

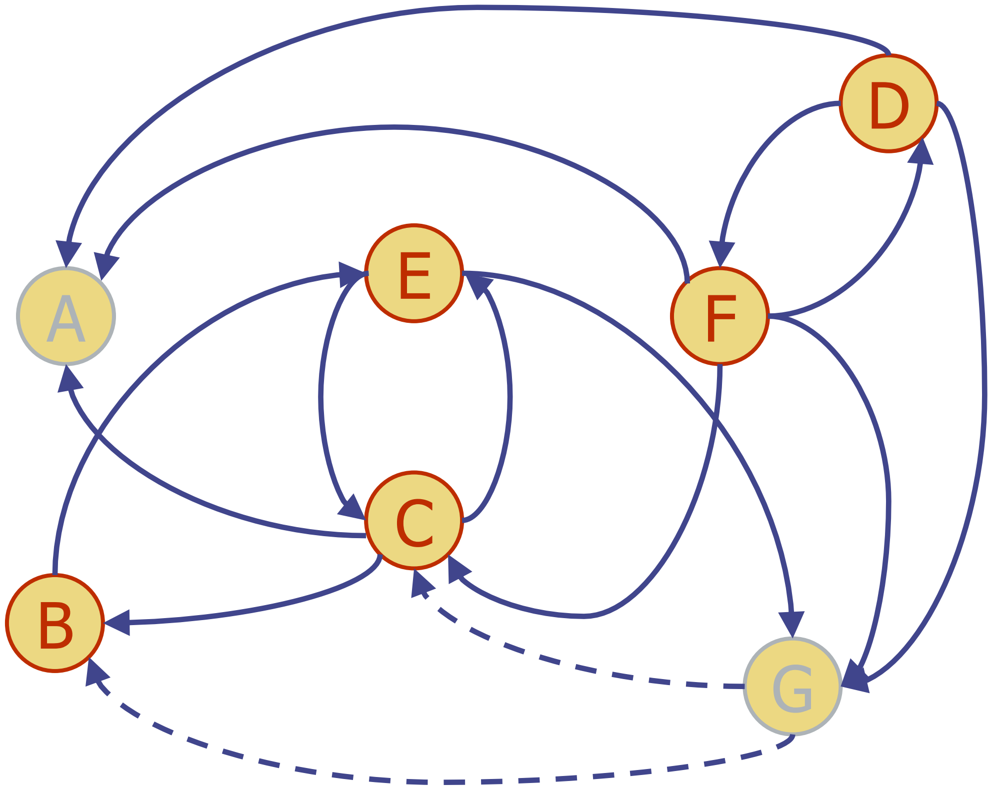
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: G
Edges to consider: to C

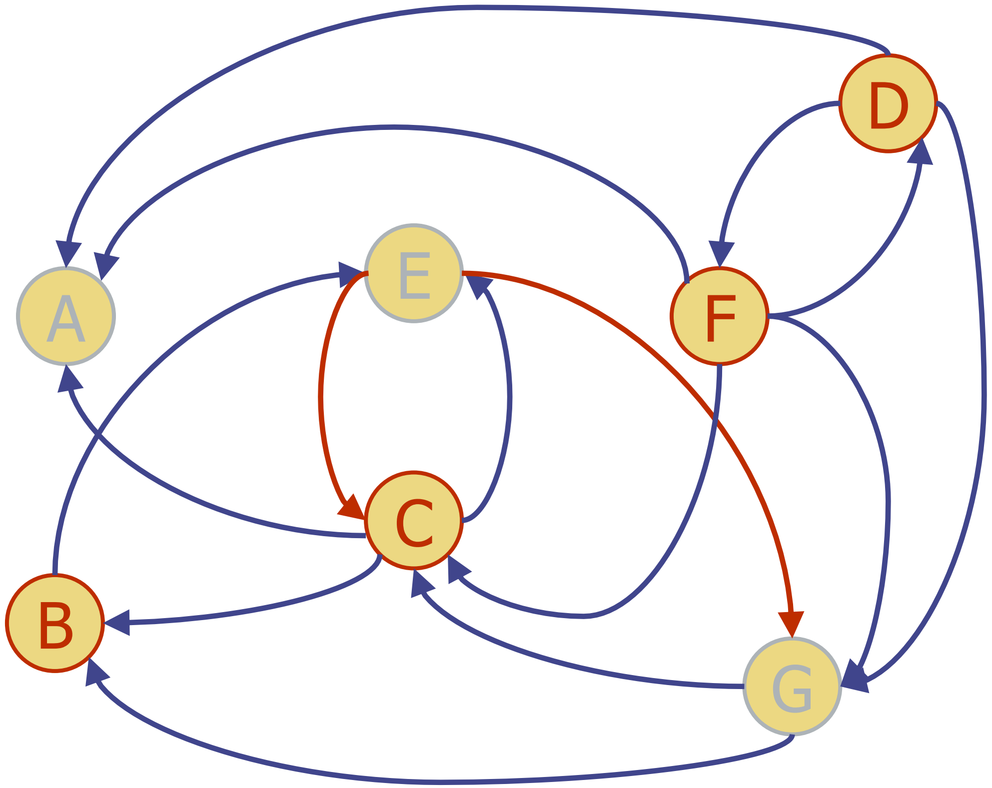
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: G
Edges to consider: -
Finished G, backtracking to E

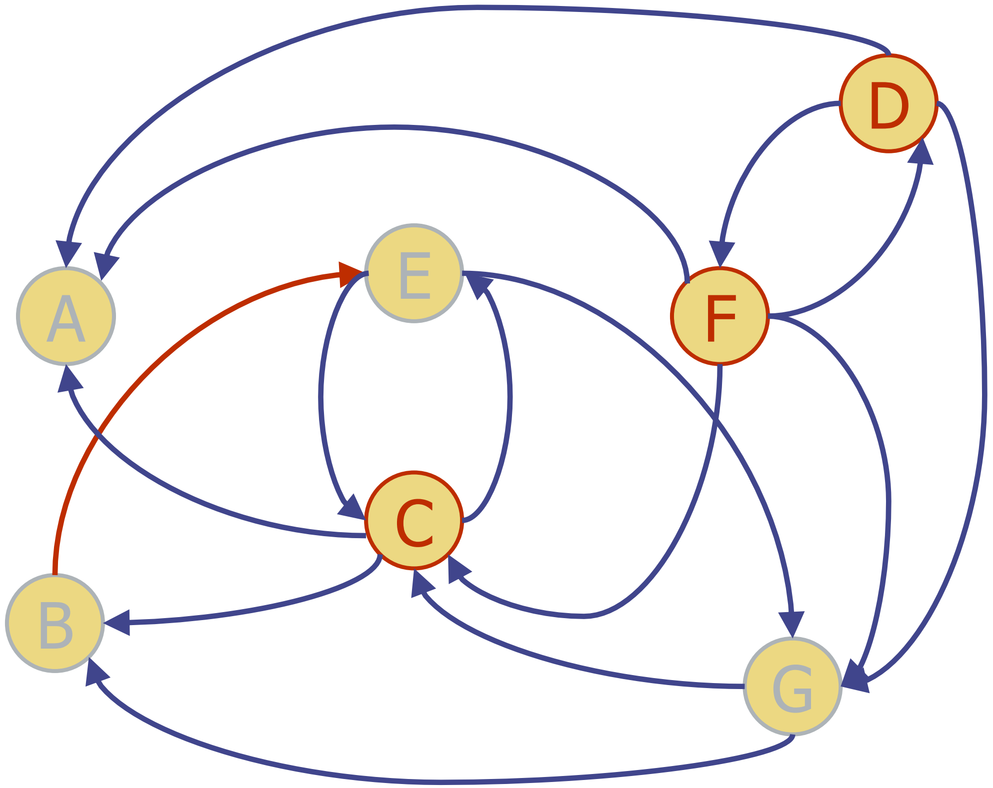
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: E
Edges to consider: -
Finished E, backtracking to B

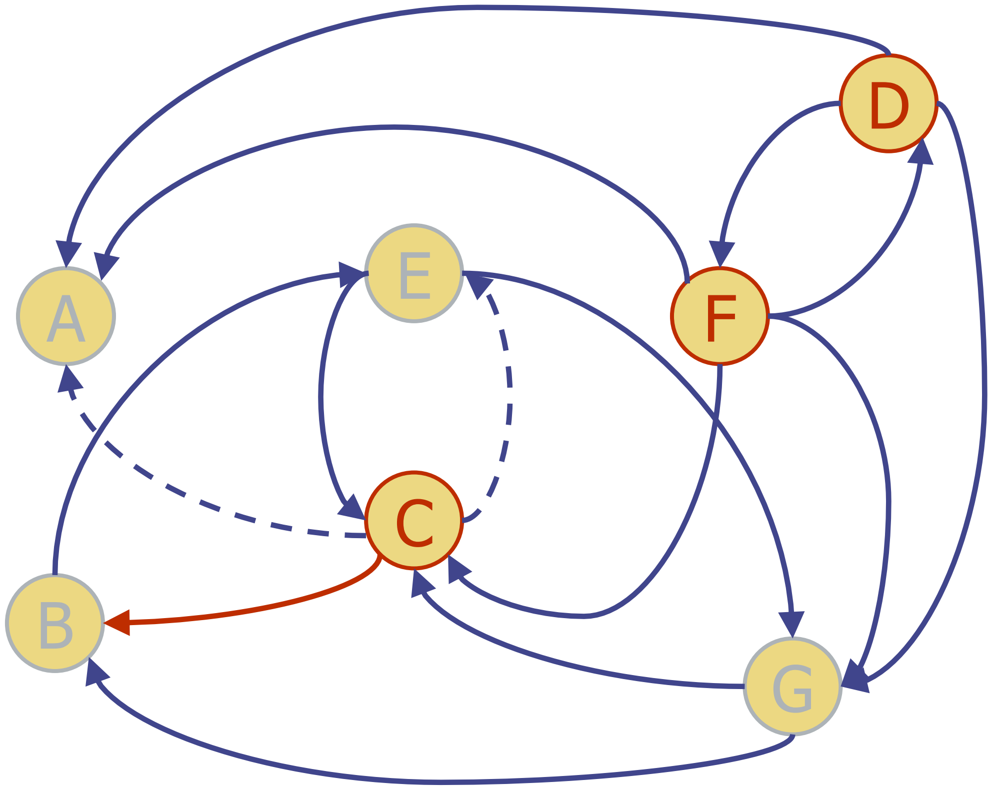
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: B
Edges to consider: -
Finished B, backtracking to C

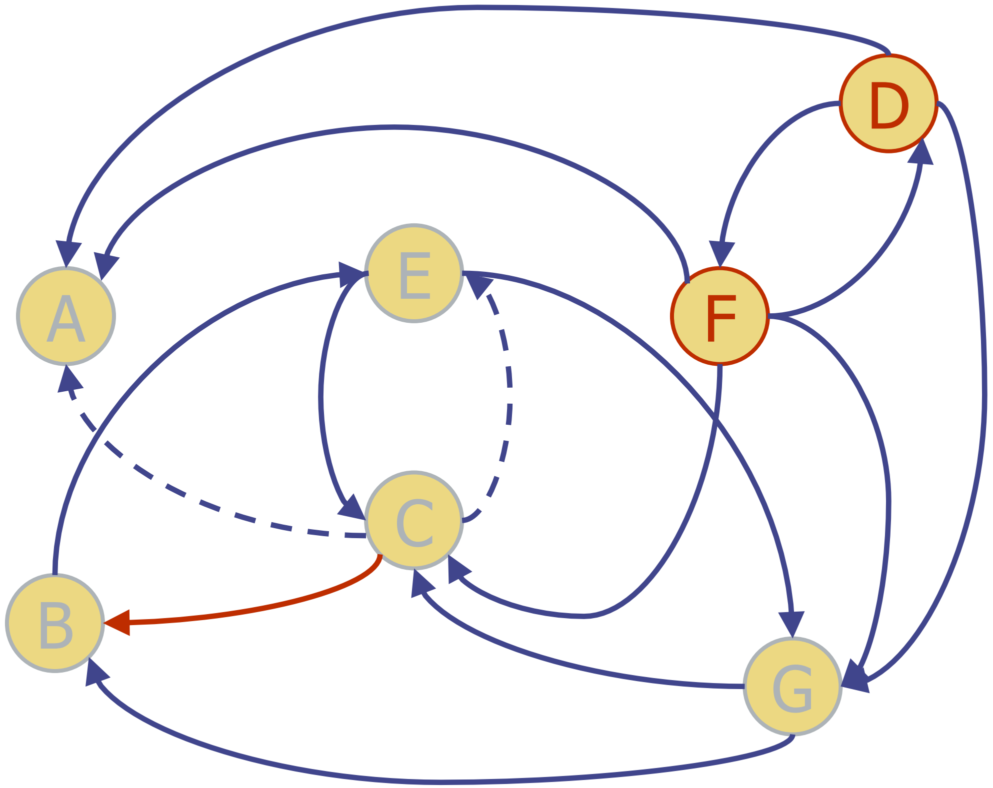
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: C
Edges to consider: E

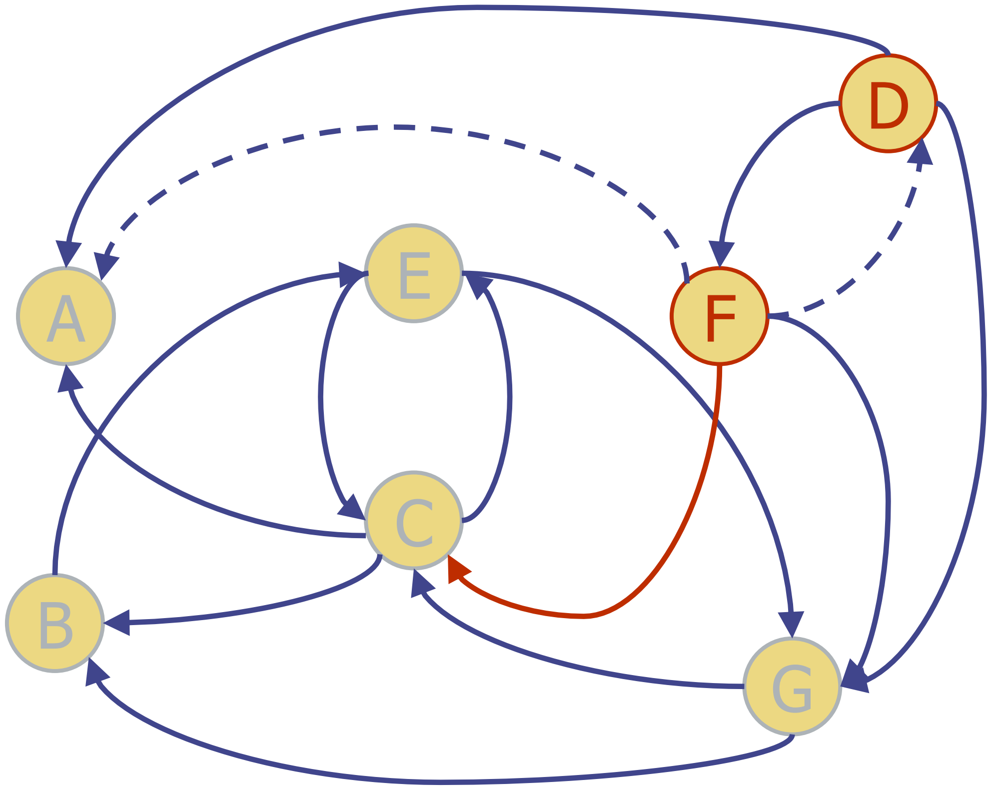
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: C
Edges to consider: -
Finished C, backtracking to F

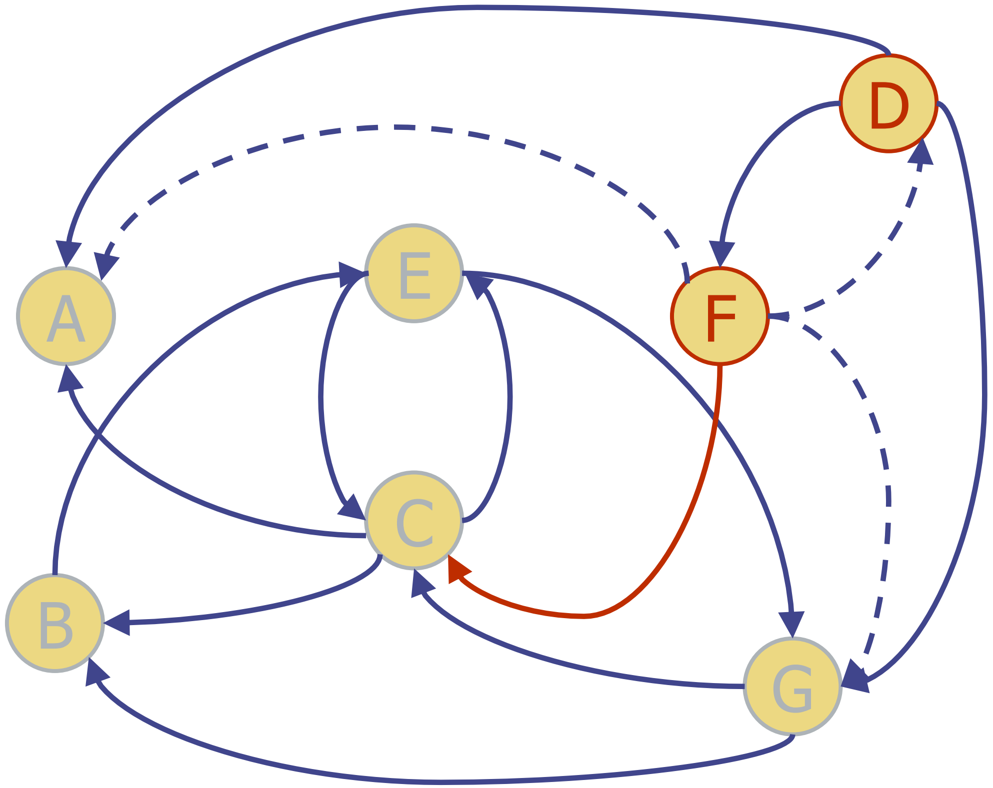
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: F
Edges to consider: to D, G

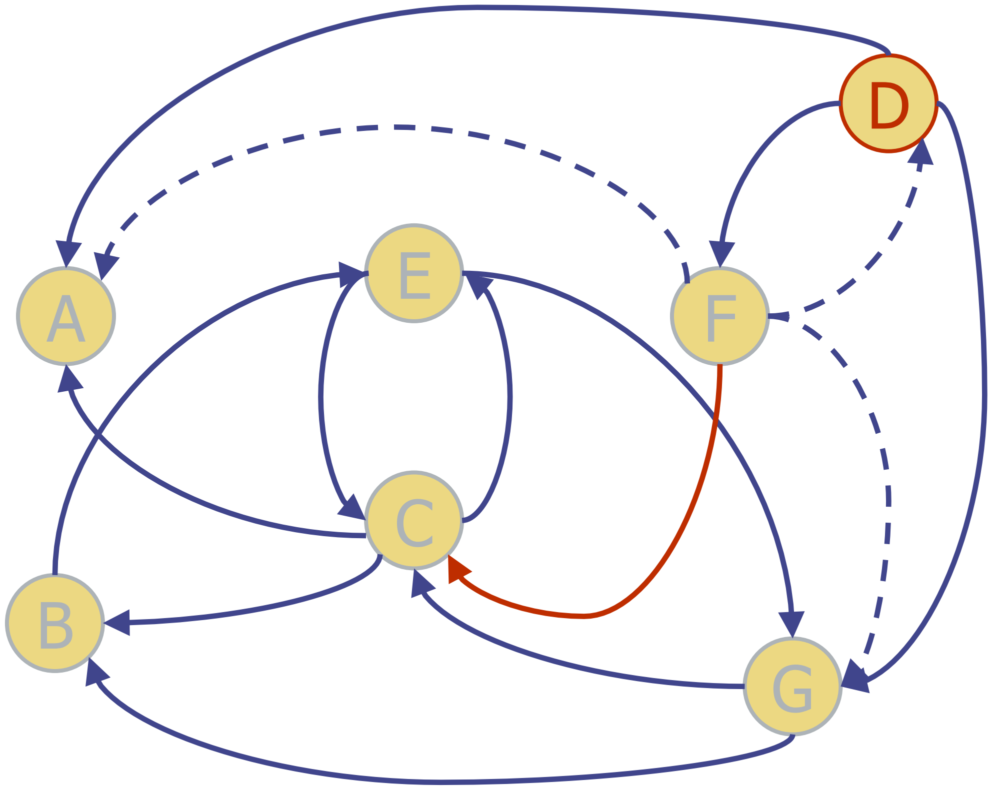
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: F
Edges to consider: to G

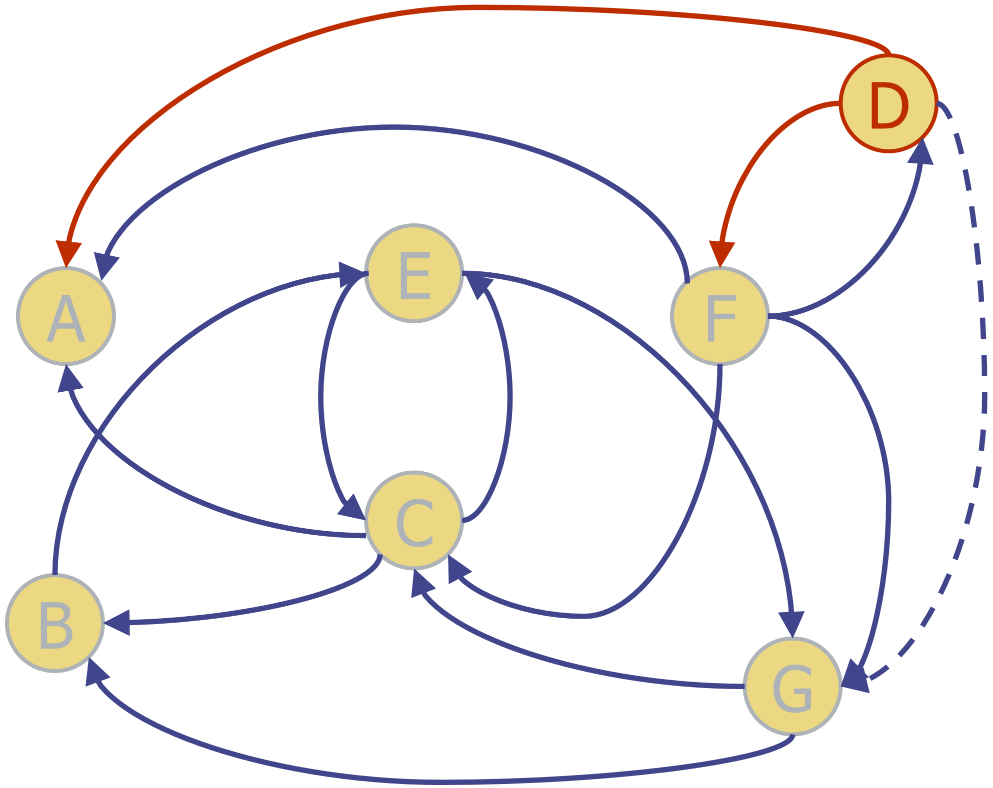
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: F
Edges to consider: -
Finished F, backtracking to D

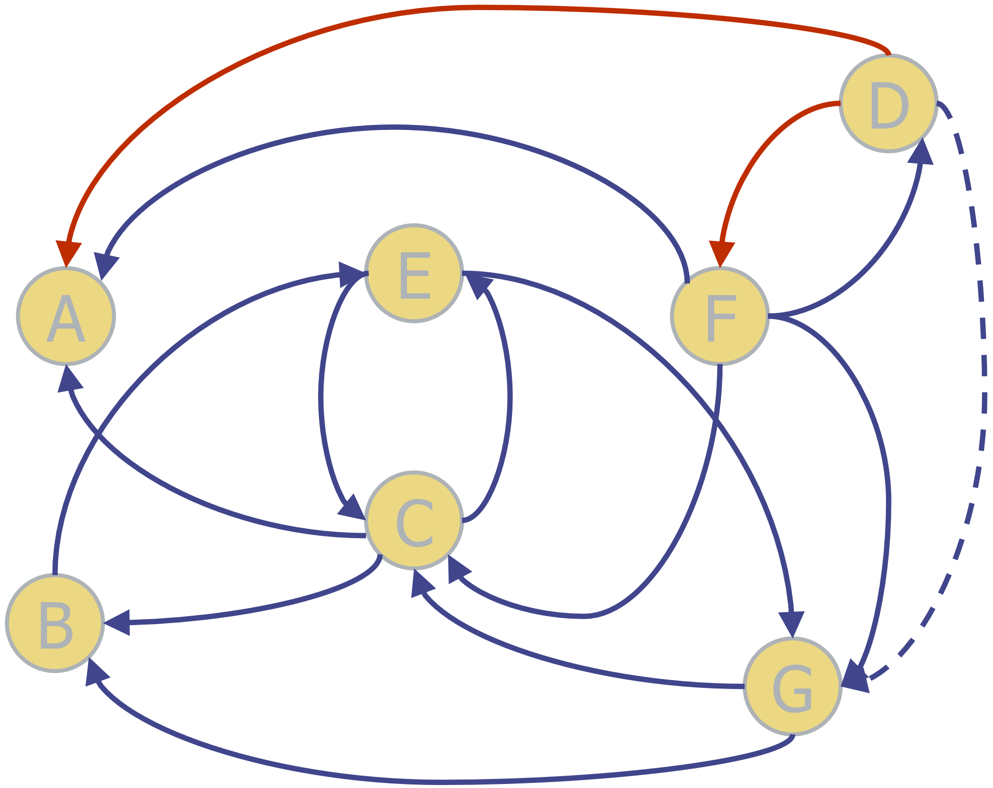
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

Current vertex: D
Edges to consider: to G

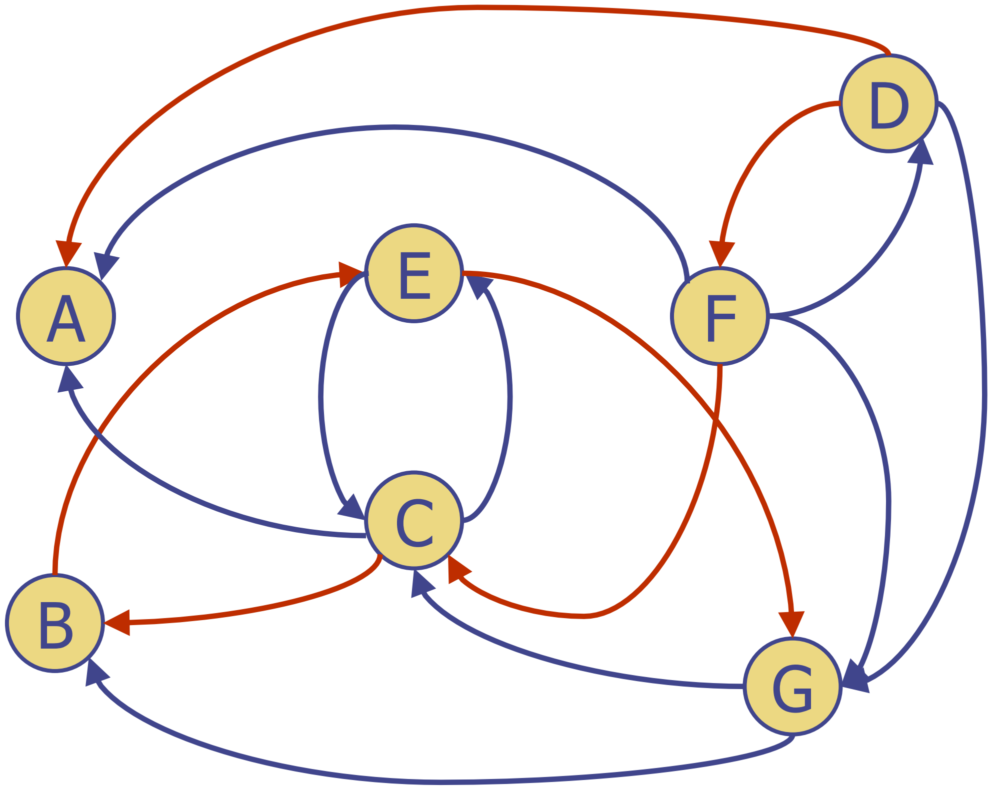
DFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

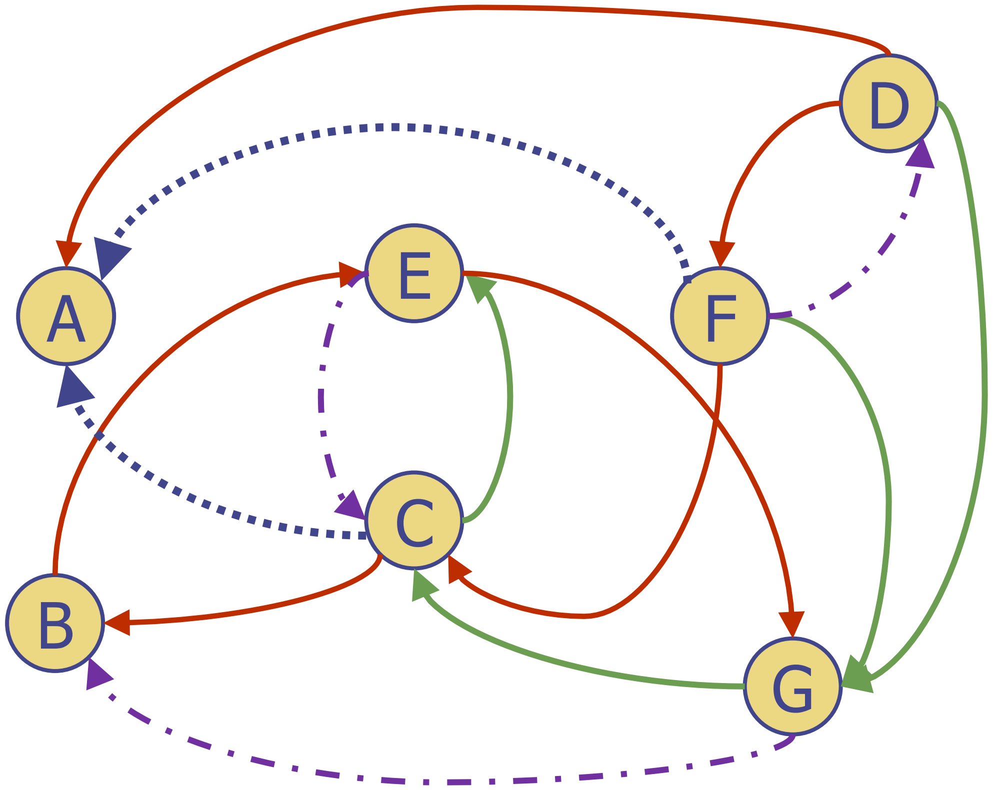
Current vertex: D
Edges to consider: -
Finished D – start vertex – stop.

DFS Traversal – discovery edges

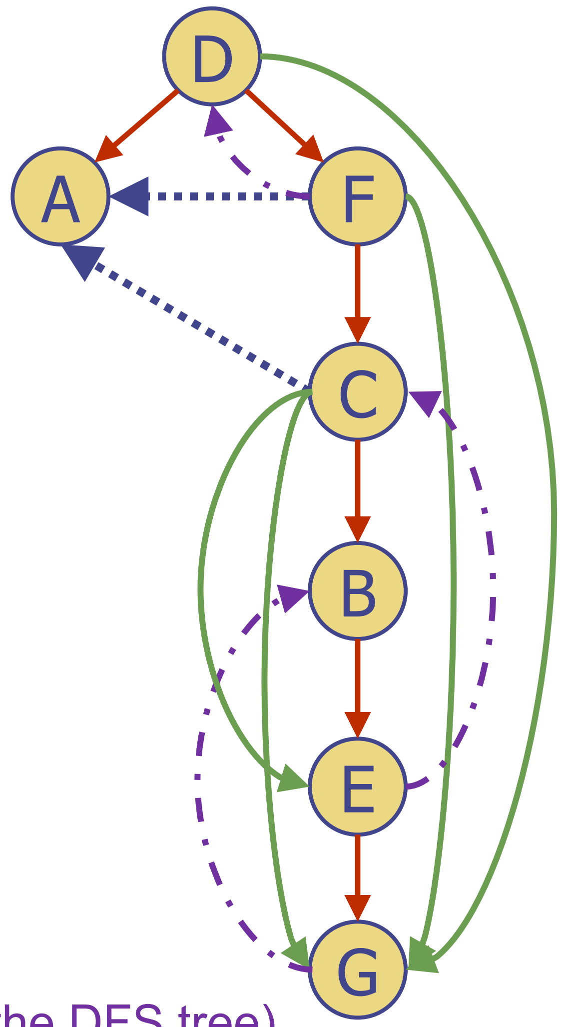


visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)

DFS Tree



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
C	(F,C)
B	(C,B)
E	(B,E)
G	(E,G)



Tree edge — discovery edge

Nontree edges:
- back edge (connects a vertex to an ancestor in the DFS tree)
- forward edge (connects a vertex to a descendant in the DFS tree)
- cross edge (connects a vertex to another vertex that is neither its ancestor nor its descendant)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Properties of a DFS in a Digraph

- Proposition. A depth-first search in a directed graph $\vec{G}$ starting at a vertex s visits all the vertices of $\vec{G}$ that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.
- Justification. Consider V_s to be the subset of vertices of $\vec{G}$ visited by a DFS starting at s. Need to show that V_s contains s and every vertex reachable from s.
  - Suppose that there is a vertex w reachable from s that is not in V_s
  - Consider a directed path from s to w and let (u, v) be the first edge on this path that goes out of $V_s \rightarrow u \in V_s, \mathrm{v} \notin V_s$
  - When DFS reaches u, all outgoing edges of u are explored – thus it must also reach v $\rightarrow$ then $v \in V_s$ (contradiction)
  - Second property – induction: each time a discovery edge (u, v) is identified, since u was previously discovered, there exists a directed path from s to u; by appending the discovery edge to the existing path, a directed path from s to v is obtained

Graph Class, part 1

```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys()

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )          # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values():
      result.update(secondary_map.values())     # add edges to resulting set
    return result
```

Graph Class, part 2

```
def get_edge(self, u, v):
  """Return the edge from u to v, or None if not adjacent."""
  return self._outgoing[u].get(v)          # returns None if v not adjacent

def degree(self, v, outgoing=True):
  """Return number of (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to count incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  return len(adj[v])

def incident_edges(self, v, outgoing=True):
  """Return all (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to request incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  for edge in adj[v].values():
    yield edge

def insert_vertex(self, x=None):
  """Insert and return a new Vertex with element x."""
  v = self.Vertex(x)
  self._outgoing[v] = { }
  if self.is_directed():
    self._incoming[v] = { }        # need distinct map for incoming edges
  return v

def insert_edge(self, u, v, x=None):
  """Insert and return a new Edge from u to v with auxiliary element x."""
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

Depth-First Search in a Directed Graph – Python Implementation

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```

```
result = {u : None}
DFS(g, u, result)
```

```
  def incident_edges(self, v, outgoing=True):
    """Return all (outgoing) edges incident to vertex v in the graph.

    If graph is directed, optional parameter used to request incoming edges.
    """
    adj = self._outgoing if outgoing else self._incoming
    for edge in adj[v].values():
      yield edge
```

DFS in a Directed Graph – Running Time

- Consider $\vec{G}$, a directed graph with n vertices and m edges. A DFS traversal of $\vec{G}$ can be performed in $O(n + m)$ time.
  - provided the graph is represented using a data structure where the incident edges of a vertex (both incoming and outgoing) can be iterated in $O(\deg(v))$ time, and finding the opposite vertex takes $O(1)$ time
  - The DFS procedure will be called at most once for every vertex of the graph
  - Each edge will be examined at most once in a directed graph, from its origin vertex

Problems Solved using a DFS Traversal in a Directed Graph

1. Computing a directed path from vertex u to vertex v, or report that no such path exists
2. Testing whether $\vec{G}$ is strongly connected
3. Computing the set of vertices of $\vec{G}$ that are reachable from a given vertex s
4. Computing a directed cycle in $\vec{G}$, or reporting that $\vec{G}$ is acyclic
5. Computing the transitive closure of $\vec{G}$

1. Compute a Directed Path from u to v

- Assume DFS was performed for the digraph
- Exactly the same algorithm as in the undirected case - build the path from end to start

```
def construct_path(u, v, discovered):
  path = [ ]                                         # empty path by default
  if v in discovered:
    # we build list from v to u and then reverse it at the end
    path.append(v)
    walk = v
    while walk is not u:
      e = discovered[walk]                           # find edge leading to walk
      parent = e.opposite(walk)
      path.append(parent)
      walk = parent
    path.reverse( )                                  # reorient path from u to v
  return path
```

2. Testing whether $\overrightarrow{G}$ is strongly connected

- That is, if for every pair of vertices u and v, u reaches v and v reaches u
- Idea: start an independent DFS traversal from each vertex of $\vec{G}$. If the `discovered` dictionary of every of these independent DFS traversals has length n (the number of vertices), then $\vec{G}$ is strongly connected
- Running time: ?

2. Testing whether $\overrightarrow{G}$ is strongly connected

- That is, if for every pair of vertices u and v, u reaches v and v reaches u
- Idea: start an independent DFS traversal from each vertex of $\vec{G}$. If the `discovered` dictionary of every of these independent DFS traversals has length n (the number of vertices), then $\vec{G}$ is strongly connected
- Running time: $O(n(n+m))$, not that great
- Better idea:
  - Start with doing a DFS from an arbitrary vertex s.
  - If the `discovered` dictionary does not contain all the vertices – the digraph is not strongly connected - stop.
  - Otherwise, construct a copy of the graph $\vec{G}$, but where the orientation of each edge is reversed. Perform a DFS on the reversed graph. If `discovered` contains all vertices – the digraph is strongly connected. Otherwise it is not.
  - Runs in $O(n+m)$ time

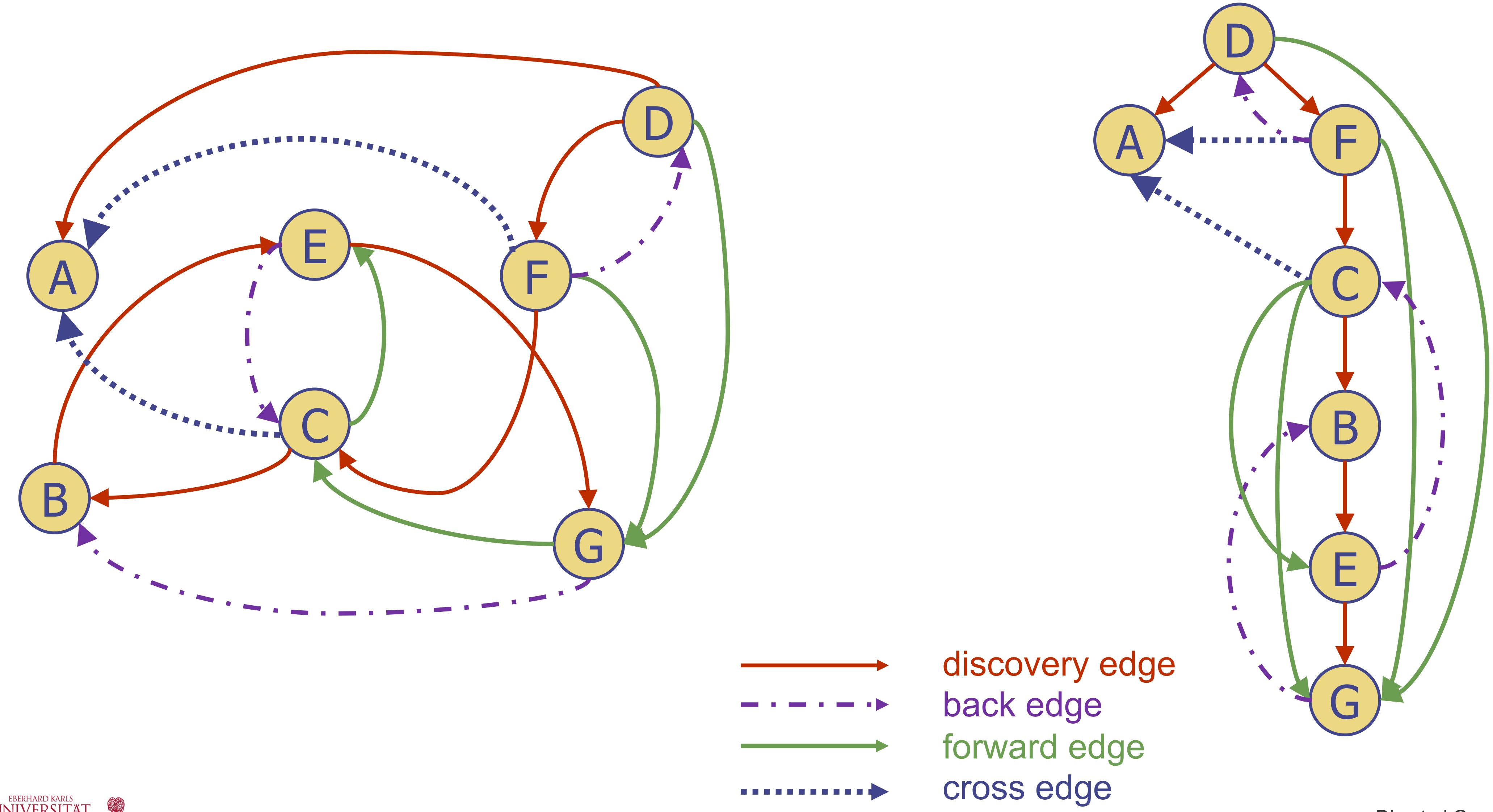
3. Computing the Vertices Reachable from a Given Start Vertex s

- Perform a DFS traversal $\vec{G}$ starting from s
- The set of vertices reachable from s are the keys of the `discovered` dictionary

4. Compute a Directed Cycle in $\vec{G}$, or Report that $\vec{G}$ is Acyclic

- The DFS procedure was already performed for the graph G
- A cycle exists if and only if a back edge exists with respect to the DFS traversal of that graph
- In a directed graph DFS traversal, there are multiple types of nontree edges: back edges, forward edges and cross edges
- When a directed edge is explored, leading to a previously visited vertex, keep track of whether that vertex is an ancestor of the current vertex
- To obtain the cycle, take the back edge from the descendant to the ancestor and then follow DFS tree edges back to the descendant

4. Compute a Directed Cycle in $\vec{G}$, or Report that $\vec{G}$ is Acyclic



5. Computing the Transitive Closure of $\vec{G}$

- Particular graph applications benefit from being able to answer reachability questions more efficiently
  - e.g. a service that computes driving destinations from point a to point b; a first step is to find about if b can be reached starting from a
- Precompute a more efficient representation for the graph that can answer such queries, and then reuse it for all the reachability queries
- A transitive closure of a directed graph $\vec{G}$ is itself a directed graph $\vec{G}^*$ such that
  - the vertices of $\vec{G}^*$ are the same vertices of $\vec{G}$ and
  - $\vec{G}^*$ has an edge (u, v) whenever $\vec{G}$ has a directed path from u to v, including the case where (u, v) is an edge of the original graph $\vec{G}$

5. Computing the Transitive Closure of $\vec{G}$: Method A

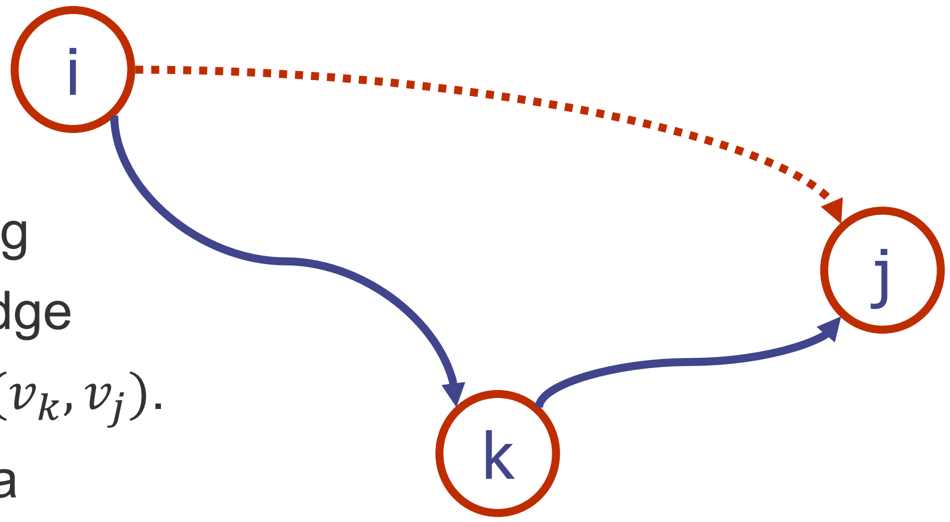
- If $\vec{G}$ is a graph with n vertices and m edges represented as an adjacency list or an adjacency map, then
- Compute the transitive closure by making n sequential DFS traversals of the graph, one starting at each vertex
- E.g. the DFS starting at vertex u will determine all vertices reachable from u – the transitive closure includes all the edges starting at u to each of the vertices that are reachable from u
- Thus computing the transitive closure of a digraph using several DFS traversals can be done in $?$ time

5. Computing the Transitive Closure of $\vec{G}$: Method A

- If $\vec{G}$ is a graph with n vertices and m edges represented as an adjacency list or an adjacency map, then
- Compute the transitive closure by making n sequential DFS traversals of the graph, one starting at each vertex
- E.g. the DFS starting at vertex u will determine all vertices reachable from u – the transitive closure includes all the edges starting at u to each of the vertices that are reachable from u
- Thus computing the transitive closure of a digraph using several DFS traversals can be done in $O(n(n+m))$ time
- Remember, the transitive closure is precomputed once and queried many times

5. Computing the Transitive Closure of $\vec{G}$: Method B

- If $\vec{G}$ is a graph with n vertices and m edges represented by a data structure that supports O(1) lookup for `get_edge(u,v)` (e.g. an adjacency matrix), then
- Compute the transitive closure of $\vec{G}$ in a series of rounds
  - Initially, $\vec{G}_0 = \vec{G}$
  - Define an arbitrary order over the vertices of $\vec{G}$, $v_1, v_2, \dots, v_n$
  - Compute the rounds, starting with round 1
  - At round k, construct a directed graph $\vec{G}_k$ starting with $\vec{G}_k = \vec{G}_{k-1}$, and adding to $\vec{G}_k$ the directed edge (v_i, v_j) if $\vec{G}_{k-1}$ contains both edges (v_i, v_k) and (v_k, v_j).
- This method of computing the transitive closure of a digraph is known as the Floyd-Warshall algorithm



Floyd-Warshall Algorithm - Pseudocode

Algorithm FloydWarshall($\vec{G}$):

Input: A directed graph $\vec{G}$ with n vertices

Output: The transitive closure $\vec{G}^*$ of $\vec{G}$

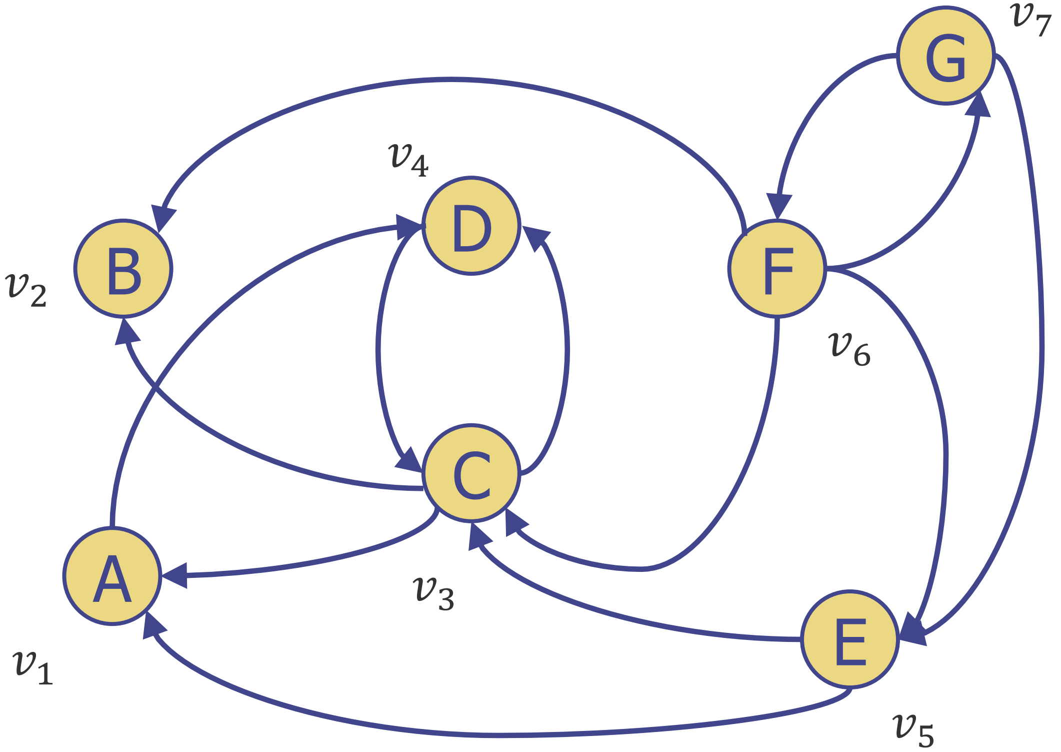
let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
$\vec{G}_0 = \vec{G}$
for $k = 1$ to n **do**
  $\vec{G}_k = \vec{G}_{k-1}$
  for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
    if both edges (v_i, v_k) and (v_k, v_j) are in $\vec{G}_{k-1}$ **then**
      add edge (v_i, v_j) to $\vec{G}_k$ (if it is not already present)
return $\vec{G}_n$

Floyd-Warshall Algorithm – Running Time

Algorithm FloydWarshall($\vec{G}$):
    Input: A directed graph $\vec{G}$ with n vertices
    Output: The transitive closure $\vec{G}^*$ of $\vec{G}$
    let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
    $\vec{G}_0 = \vec{G}$
    for $k = 1$ to n **do**
        $\vec{G}_k = \vec{G}_{k-1}$
        for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
            if both edges (v_i, v_k) and (v_k, v_j) are in $\vec{G}_{k-1}$ **then**
                add edge (v_i, v_j) to $\vec{G}_k$ (if it is not already present)
    return $\vec{G}_n$

- If the data structure supports `get_edge` and `insert_edge` in O(1) time
- The main loop, indexed by k, is executed n times
- The inner loop contains of $O(n^2)$ pairs of vertices, for each of which an $O(1)$ computation is performed
- Total running time of the algorithm: $O(n^3)$
- Asymptotically, this is not better than running DFS n times, which is $O(n^2 + nm)$
- Floyd-Warshall matches the asymptotic bounds of repeated DFS when the graph is dense

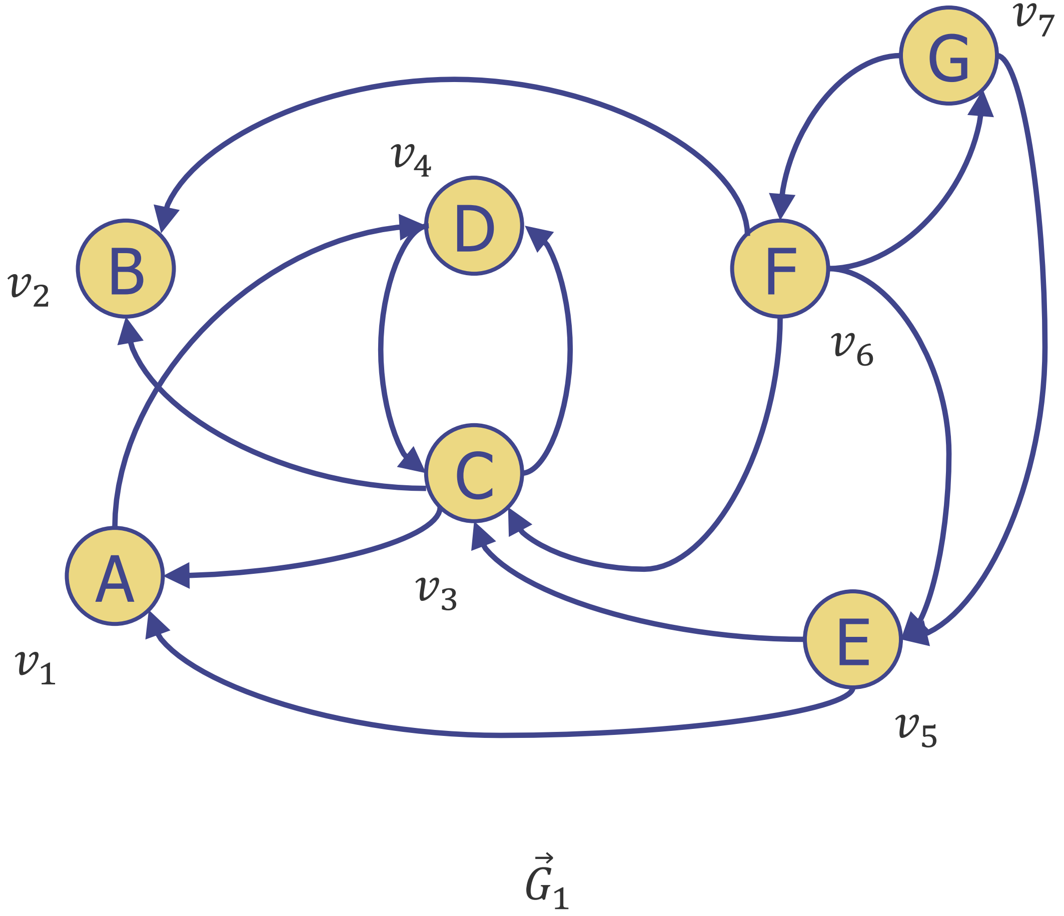
Floyd-Warshall Algorithm - Example



$$\vec{G} = \vec{G}_0$$

i	k	j

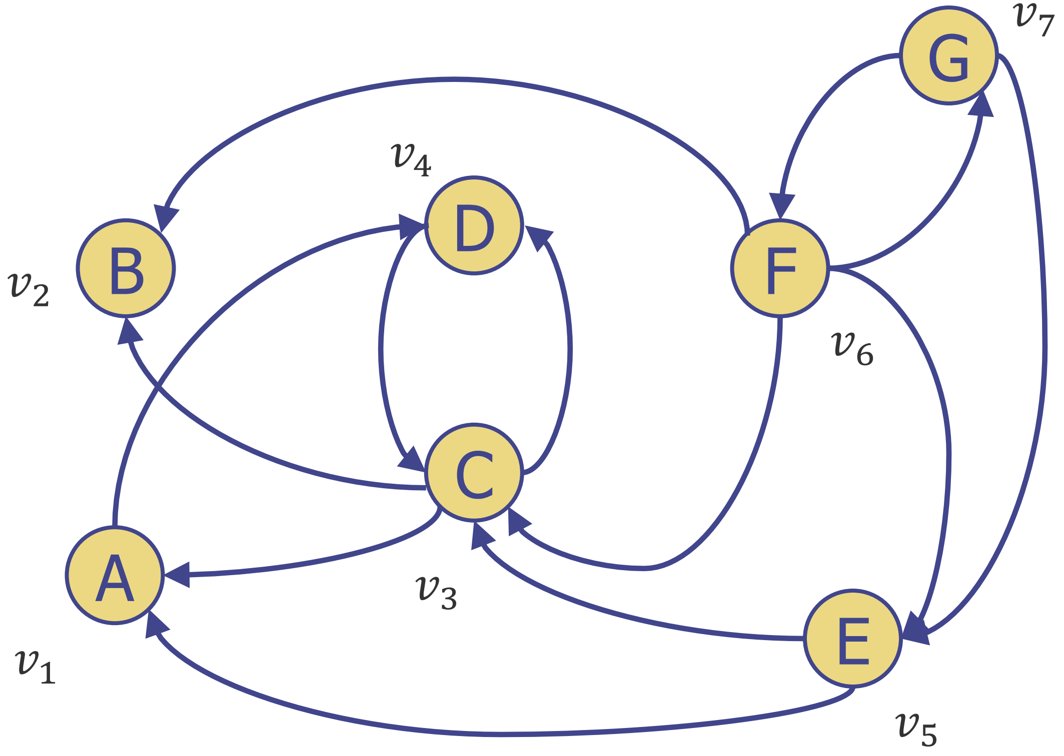
Floyd-Warshall Algorithm - Example



i	k	j
1	1	1

i=j, continue

Floyd-Warshall Algorithm - Example

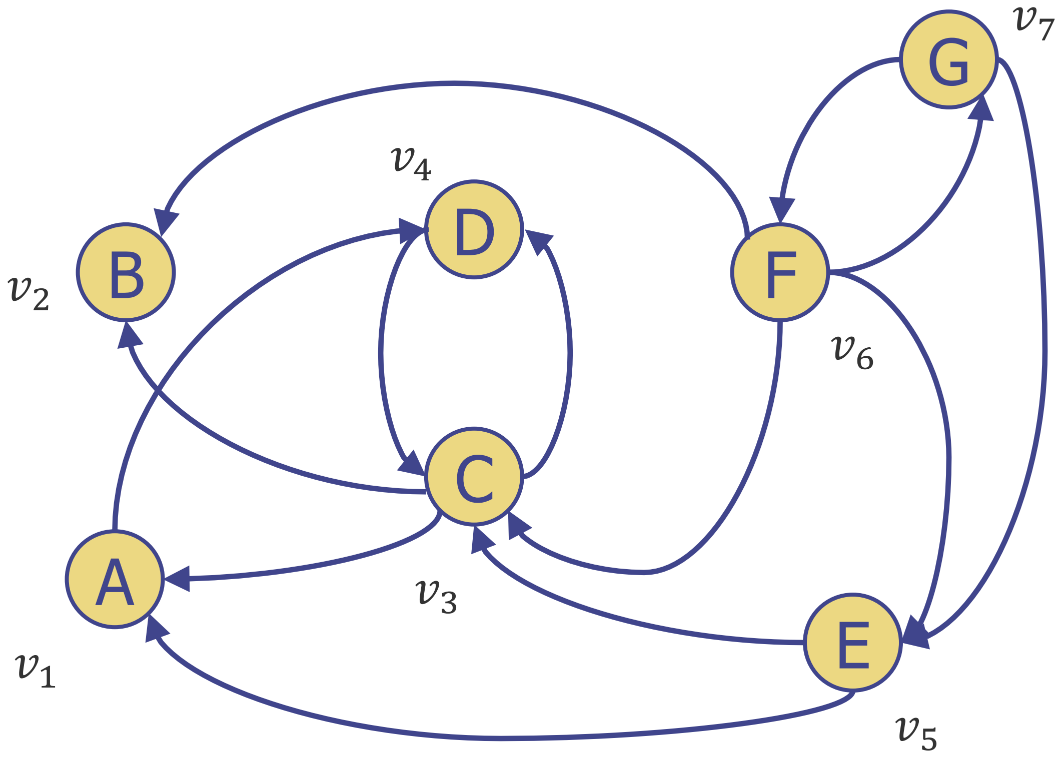


$\vec{G}_1$

i	k	j
1	1	1
2		

j=k, continue

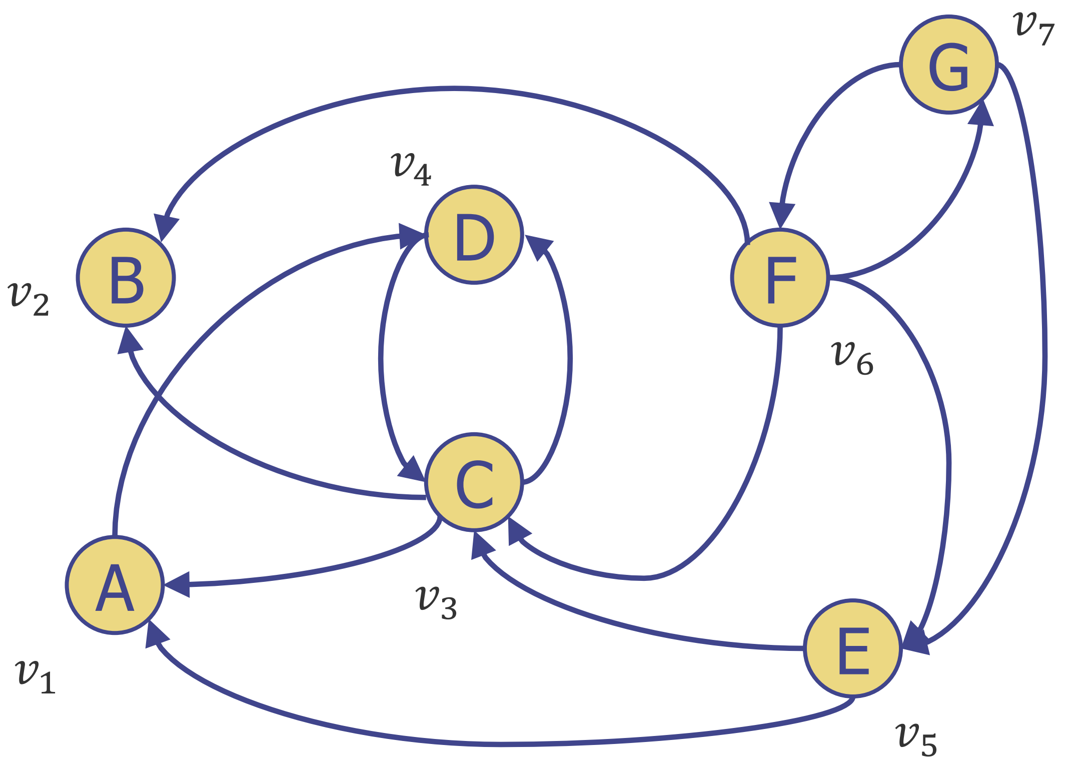
Floyd-Warshall Algorithm - Example



$\vec{G}_1$

i	k	j	
1	1	1	
2		2	i=j, continue

Floyd-Warshall Algorithm - Example



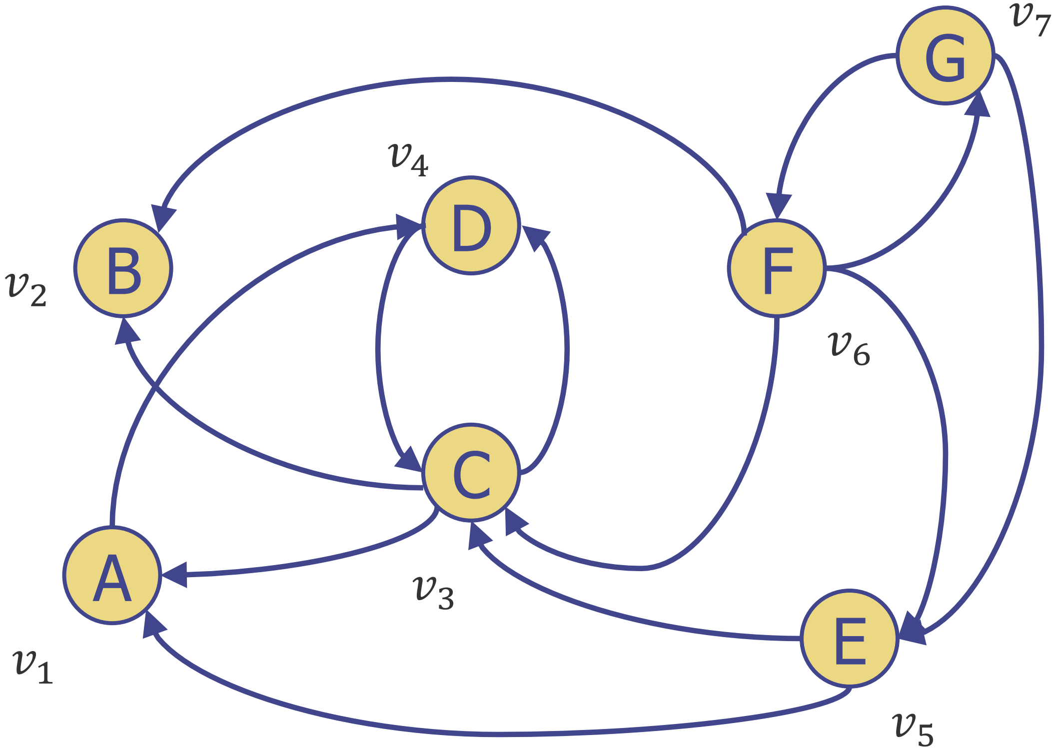
$\vec{G}_1$

i	k	j
1	1	1
2		2
		3

$i \neq j \neq k$

Does $\vec{G}$ have the edges (v_2, v_1) and (v_1, v_3)? No, it doesn't have either, continue.

Floyd-Warshall Algorithm - Example



$\vec{G}_1$

i	k	j
1	1	1
2		2
		3
		4

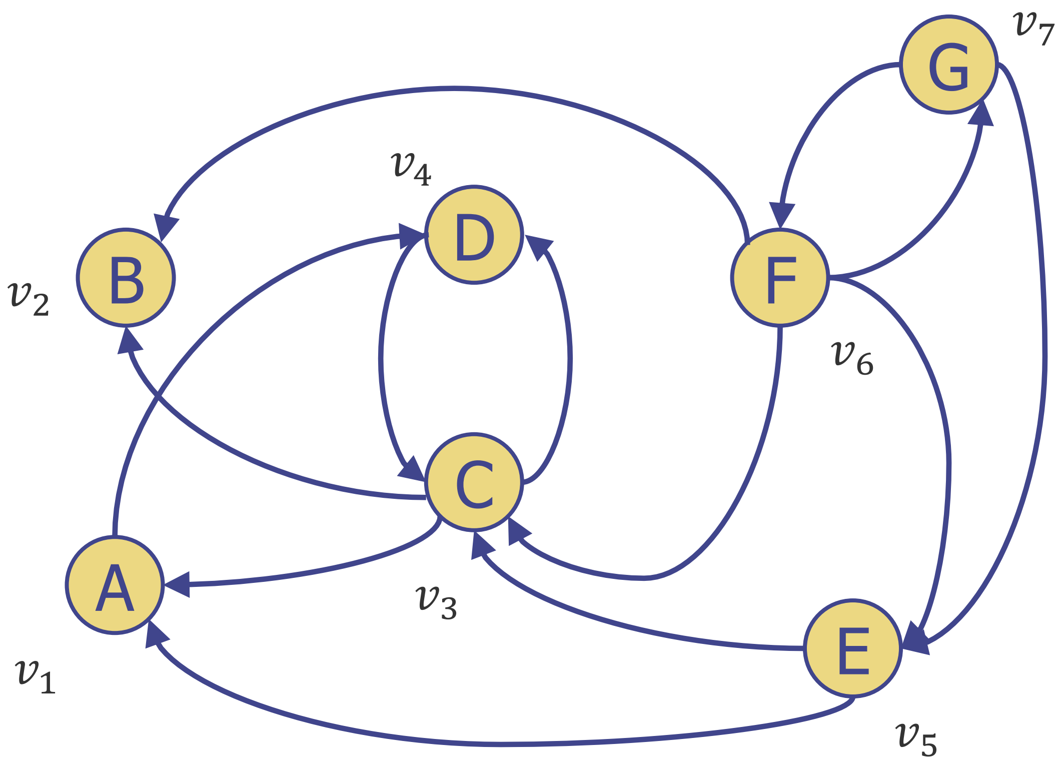
$i \neq j \neq k$

Does $\vec{G}$ have:

- (v_2, v_1) - no
- (v_1, v_4) – yes

Continue.

Floyd-Warshall Algorithm - Example



$\vec{G}_1$

i	k	j
1	1	1
2		2
		3
		4
		5

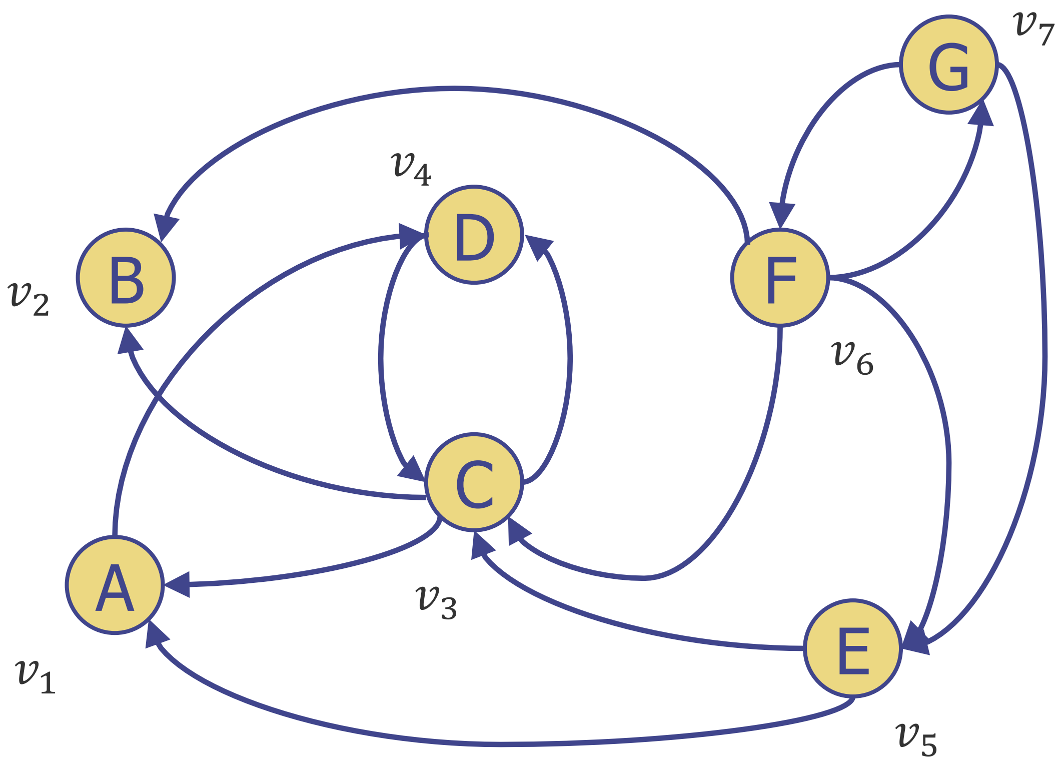
$i \neq j \neq k$

Does $\vec{G}$ have:
- (v_2, v_1) - no
- (v_1, v_5) - no

Continue.

2-1-6, 2-1-7 will also not add edges because (v_2, v_1) does not exist.

Floyd-Warshall Algorithm - Example



$\vec{G}_1$

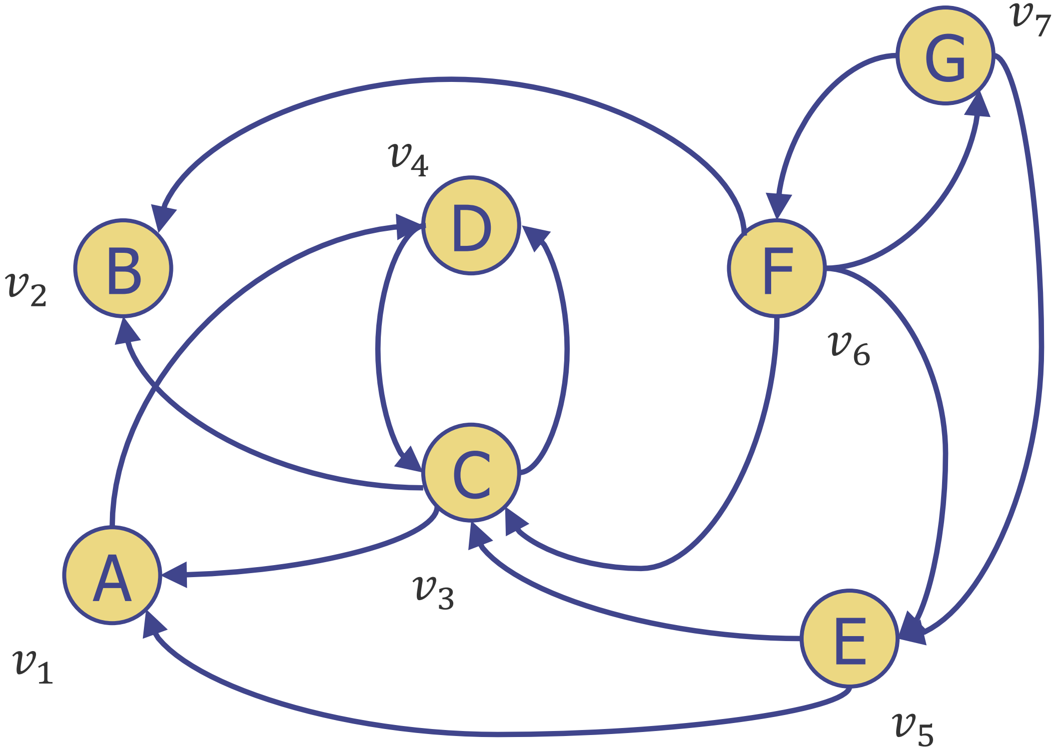
i	k	j
1	1	1
2		2
3		

$i \neq j \neq k$

Does $\vec{G}$ have:
- (v_3, v_1) - yes
- (v_1, v_2) - no

Continue.

Floyd-Warshall Algorithm - Example

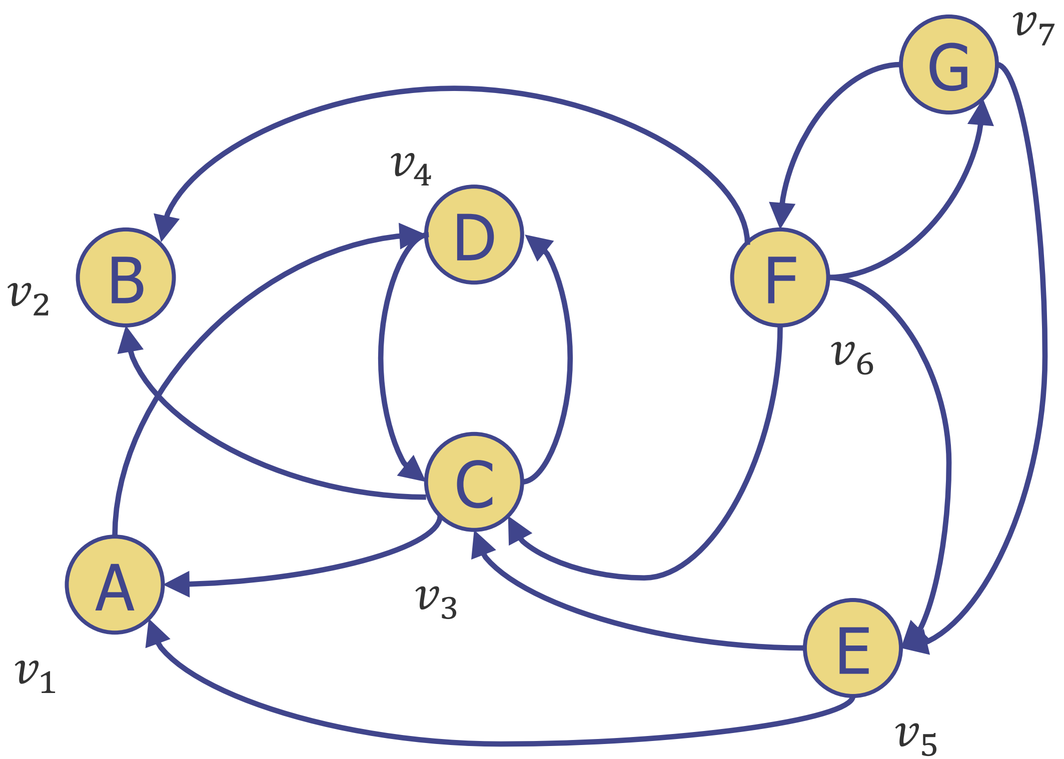


$\vec{G}_1$

i	k	j
1	1	1
2		2
3		3

$i = j$,
continue

Floyd-Warshall Algorithm - Example



$\vec{G}_1$

i	k	j
1	1	1
2		2
3		3
		4

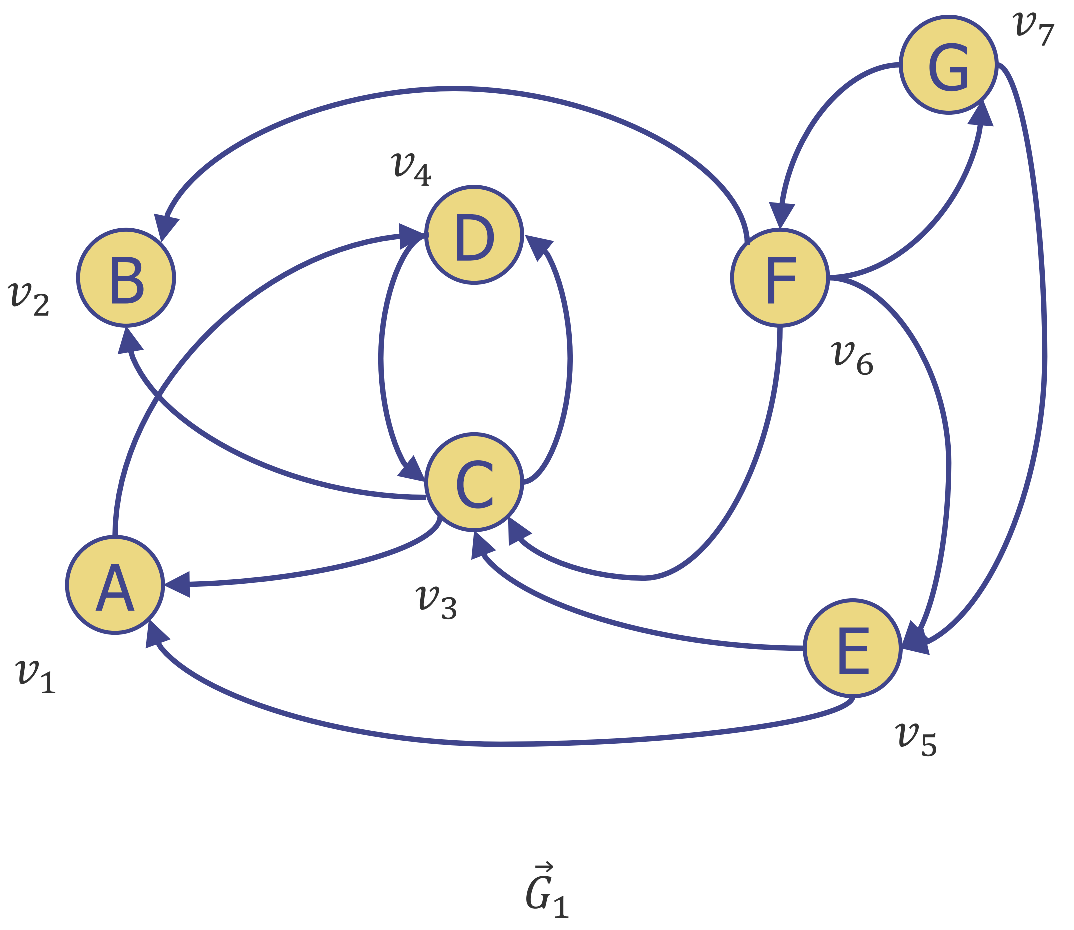
$i \neq j \neq k$

Does $\vec{G}$ have:

- (v_3, v_1) - yes
- (v_1, v_4) - yes

A direct edge (v_3, v_4) already exists, continue.

Floyd-Warshall Algorithm - Example



i	k	j
1	1	1
2		2
3		3
		4
		5

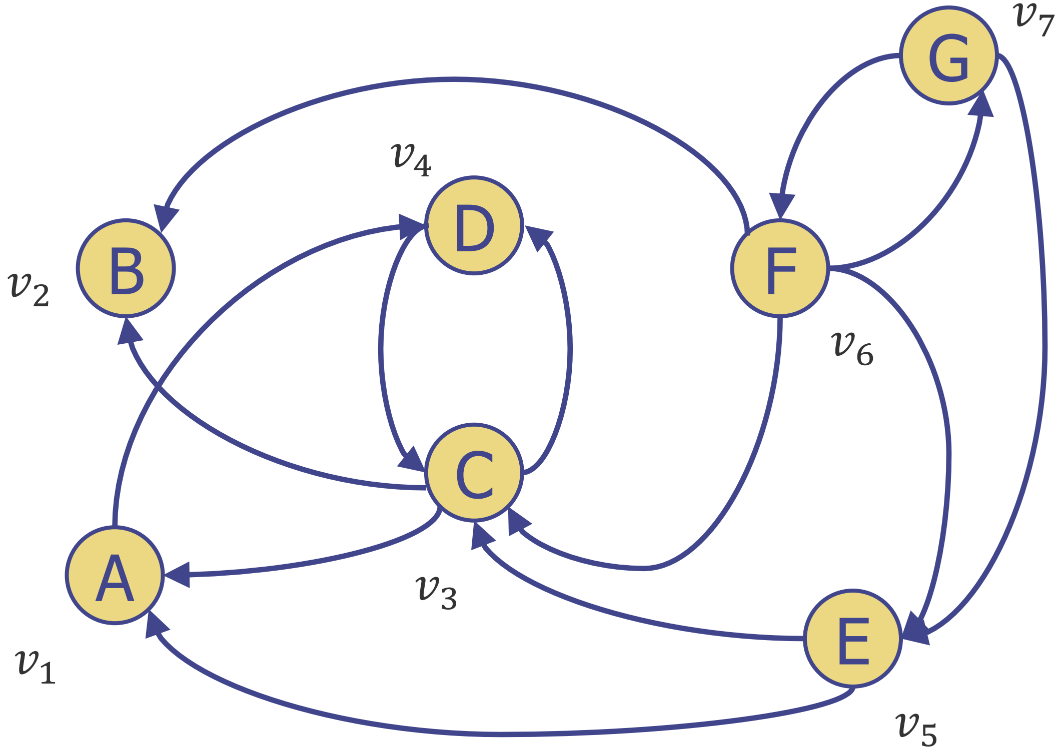
$i \neq j \neq k$

Does $\vec{G}$ have:

- (v_3, v_1) - yes
- (v_1, v_5) - no

Continue. 3-1-6 and 3-1-7 wont add edges, since 1-6 and 1-7 do not exist.

Floyd-Warshall Algorithm - Example

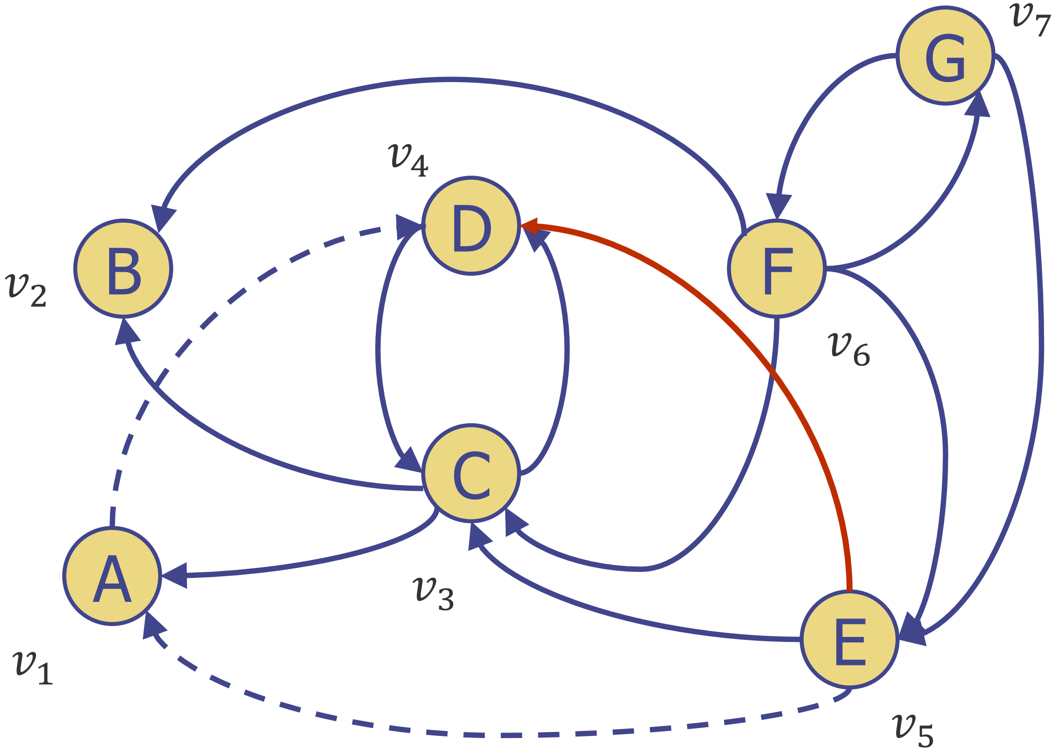


$\vec{G}_1$

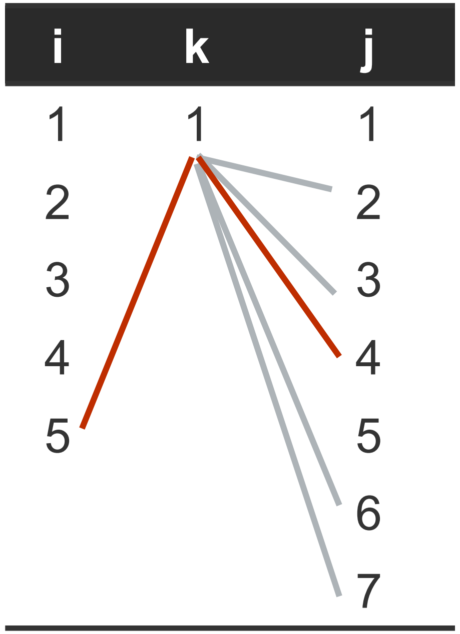
i	k	j
1	1	1
2		2
3		3
4		4
		5
		6
		7

No edges added starting with 4-1.

Floyd-Warshall Algorithm - Example

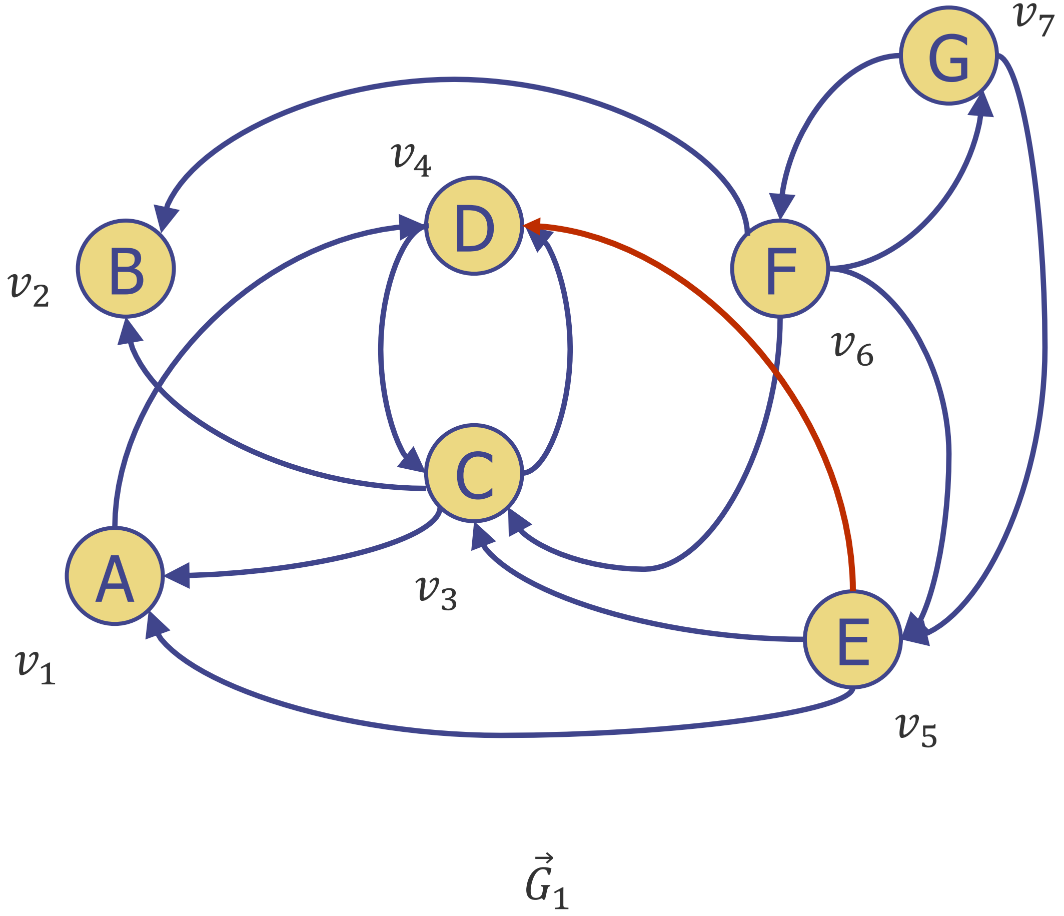


$\vec{G}_1$



The edge 5-1-4 can be added, a direct edge from 5 to 4 does not exist.

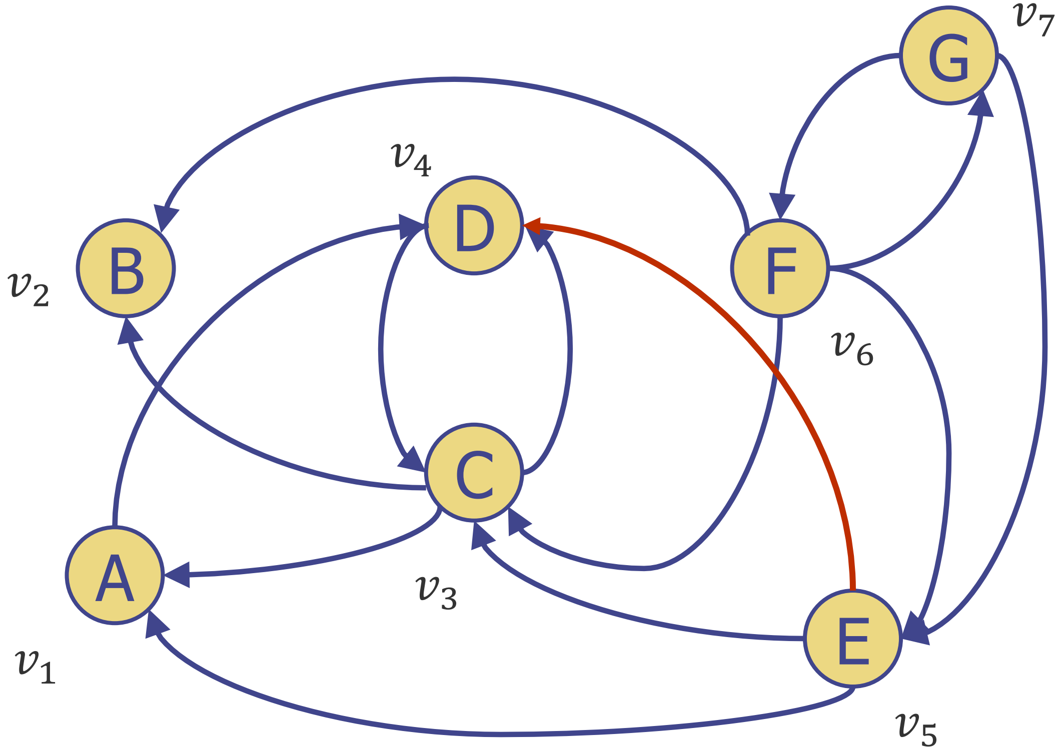
Floyd-Warshall Algorithm - Example



i	k	j
1	1	1
2		2
3		3
4		4
5		5
6		6
		7

No edges added starting with 6-1.

Floyd-Warshall Algorithm - Example

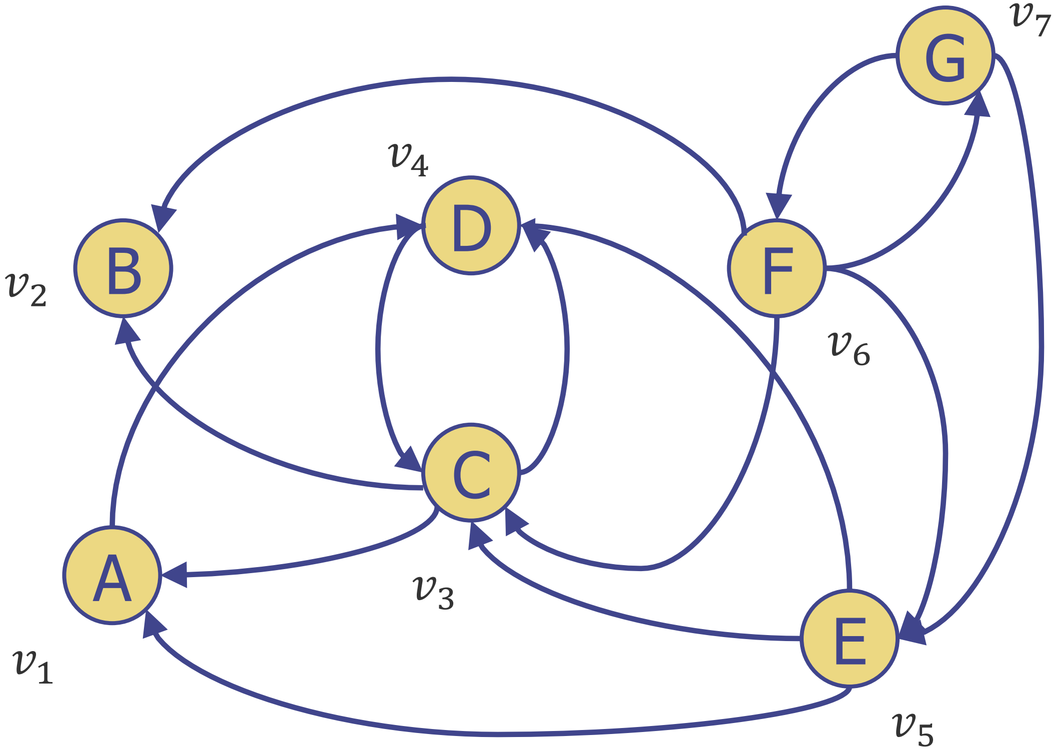


$\vec{G}_1$

i	k	j
1	1	1
2		2
3		3
4		4
5		5
6		6
7		7

No edges added starting with 7-1.

Floyd-Warshall Algorithm - Example

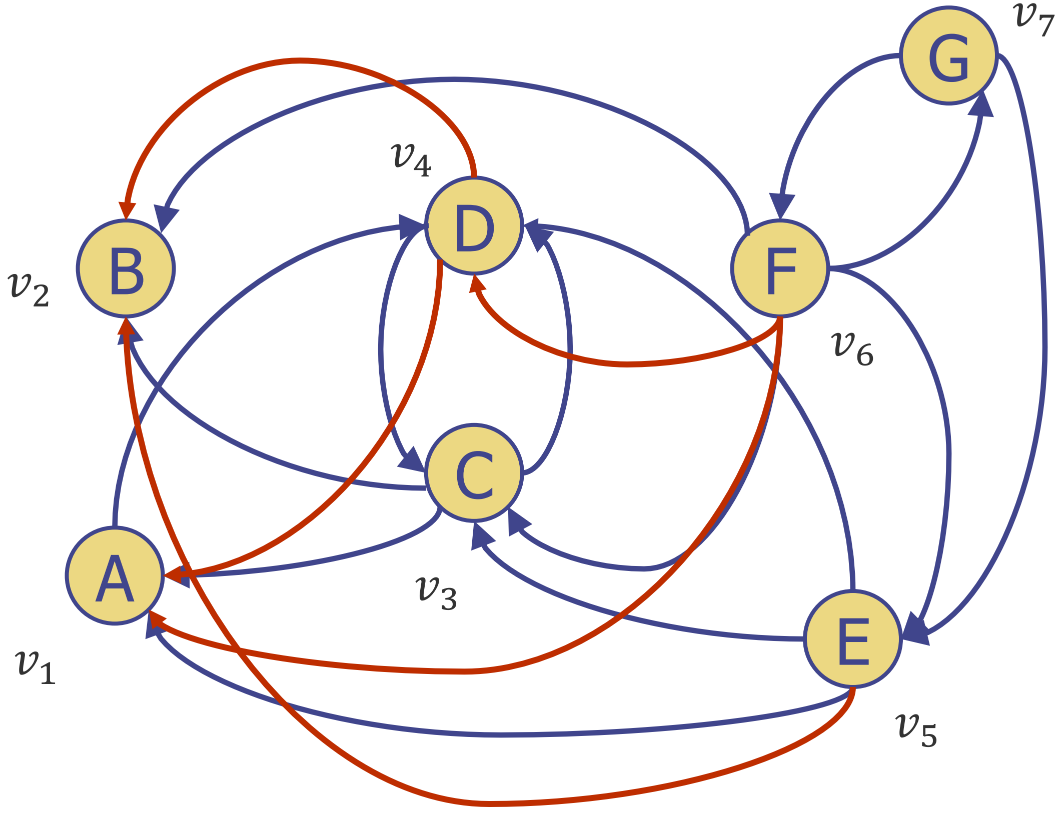


$\vec{G}_2$

i	k	j
1	1	1
	2	2
		3
		4
		5
		6
		7

The are no outgoing edges for v_2 - so no edges are added to the transitive closure.

Floyd-Warshall Algorithm - Example

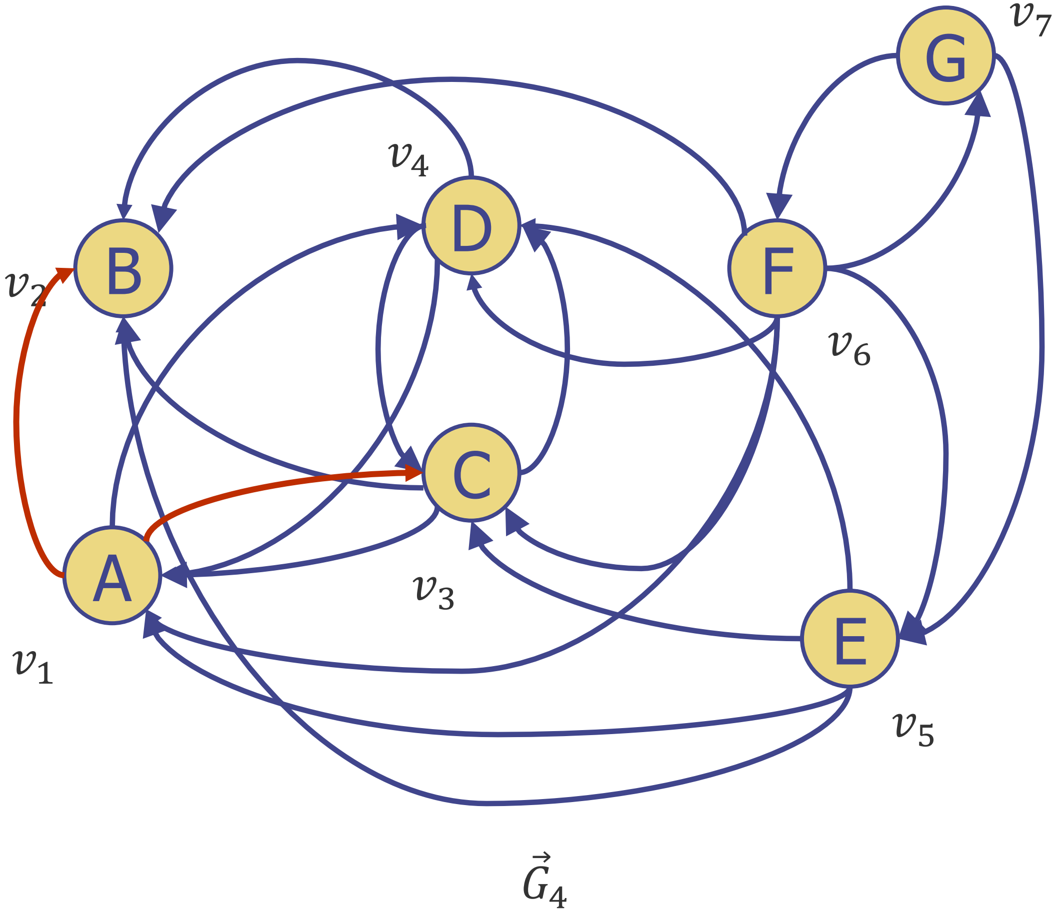


$\vec{G}_3$

i	k	j
1	1	1
2	2	2
3	3	3
4		4
5		5
6		6
7		7

4-3-1, add edge from 4 to 1
4-3-2, add edge from 4 to 2
5-3-1, a direct 5-1 edge exists
5-3-2, add edge from 5 to 2
5-3-4, a direct 5-4 edge exists
6-3-1, add edge from 6 to 1
6-3-2, a direct 6-2 edge exists
6-3-4, add edge from 6 to 4

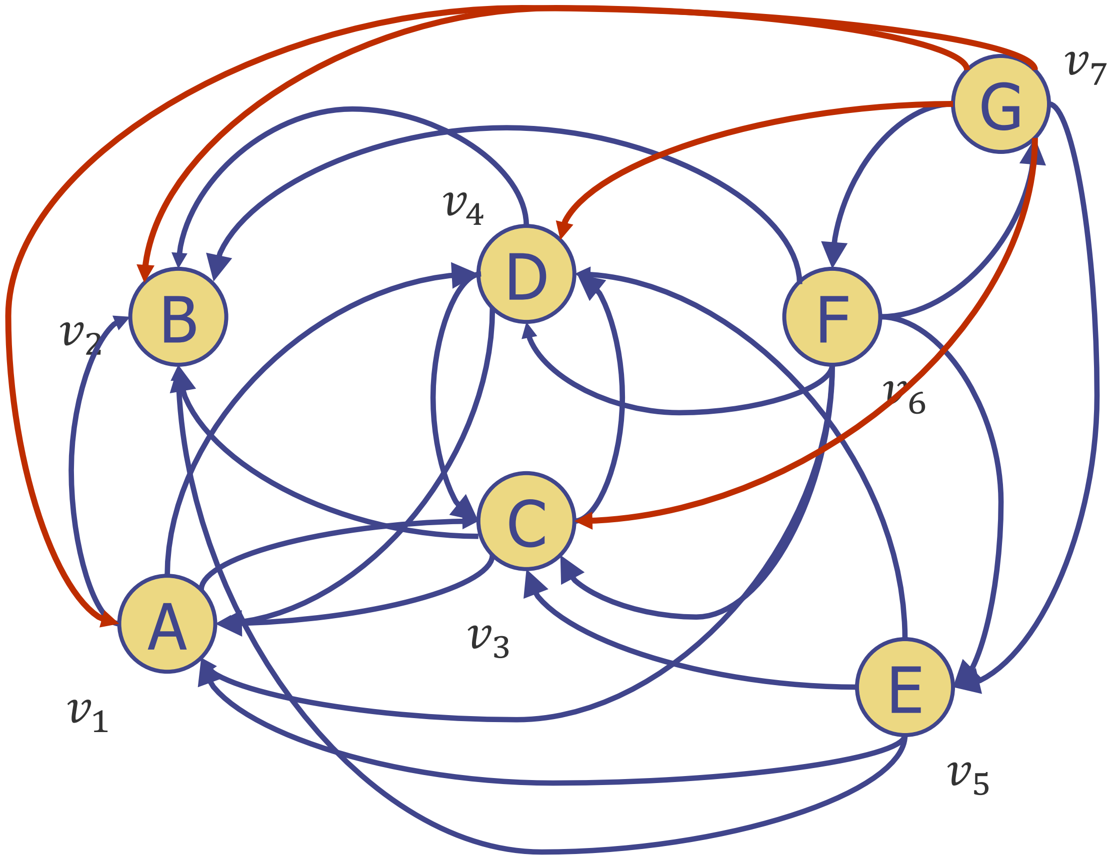
Floyd-Warshall Algorithm - Example



i	k	j
1	1	1
2	2	2
3	3	3
4	4	4
5		5
6		6
7		7

1-4-2, add edge from 1 to 2
1-4-3, add edge from 1 to 3
3-4-1, a direct 3-1 edge exists
3-4-2, a direct 3-2 edge exists
5-4-1, a direct 5-1 edge exists
5-4-2, a direct 5-2 edge exists
5-4-3, a direct 5-3 edge exists
6-4-1, a direct 6-1 edge exists
6-4-2, a direct 6-2 edge exists
6-4-3, a direct 6-3 edge exists

Floyd-Warshall Algorithm - Example



$\vec{G}_5$

i	k	j
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6		6
7		7

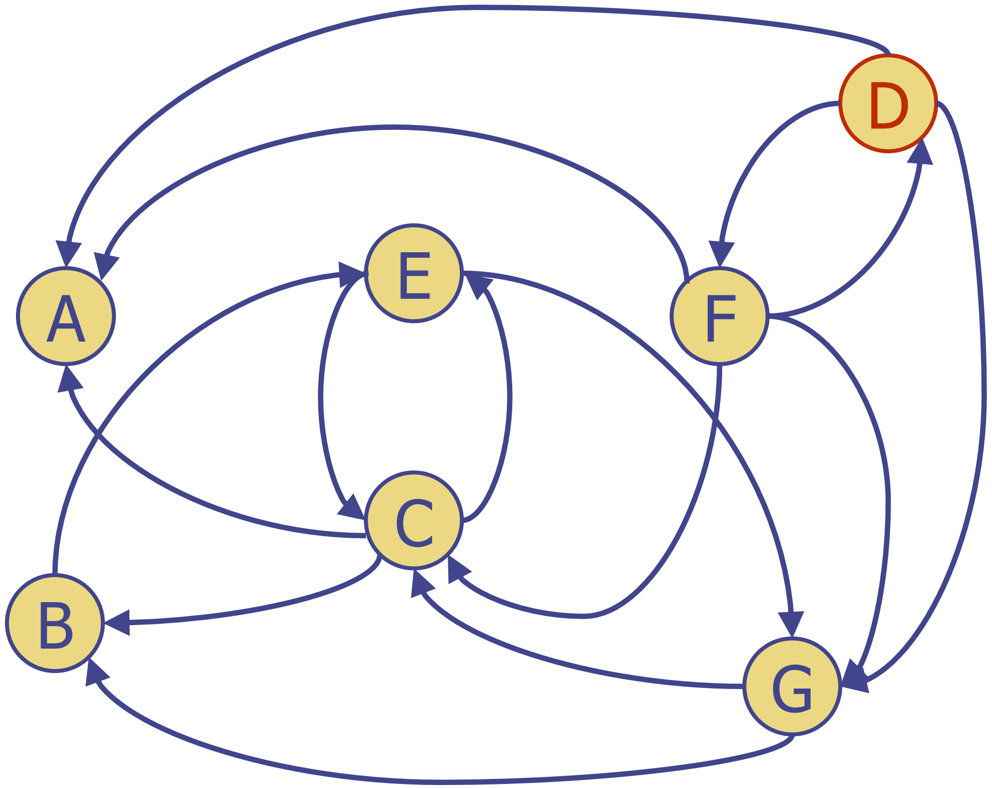
6-5-1, a direct 6-1 edge exists
6-5-2, a direct 6-2 edge exists
6-5-3, a direct 6-3 edge exists
6-5-4, a direct 6-4 edge exists
7-5-1, add edge from 7 to 1
7-5-2, add edge from 7 to 2
7-5-3, add edge from 7 to 3
7-5-4, add edge from 7 to 4
$\vec{G}_5 = \vec{G}_6 = \vec{G}_7$, stop.

Floyd-Warshall Algorithm – Python Implementation

```
def floyd_warshall(g):
  """Return a new graph that is the transitive closure of g."""
  closure = deepcopy(g)                        # imported from copy module
  verts = list(closure.vertices())             # make indexable list
  n = len(verts)
  for k in range(n):
    for i in range(n):
      # verify that edge (i,k) exists in the partial closure
      if i != k and closure.get_edge(verts[i],verts[k]) is not None:
        for j in range(n):
          # verify that edge (k,j) exists in the partial closure
          if i != j != k and closure.get_edge(verts[k],verts[j]) is not None:
            # if (i,j) not yet included, add it to the closure
            if closure.get_edge(verts[i],verts[j]) is None:
              closure.insert_edge(verts[i],verts[j])
  return closure
```

BFS in a Directed Graph

BFS in a Directed Graph - Example

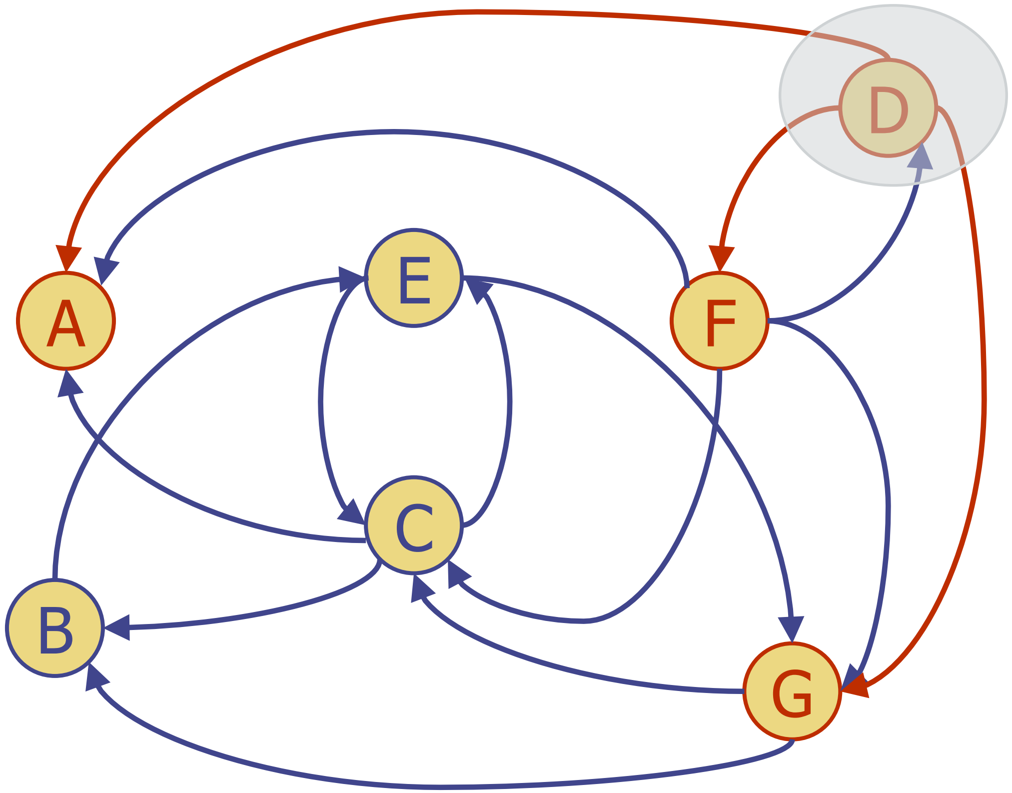


visited	discovery edge
D	None

Current vertex: D
Edges to consider: to A, F, G

- Start from vertex D, which is marked as visited (red)
- Assume that the outgoing edges of a vertex are considered in alphabetical order – e.g. for D: A, F, G

BFS in a Directed Graph - Example



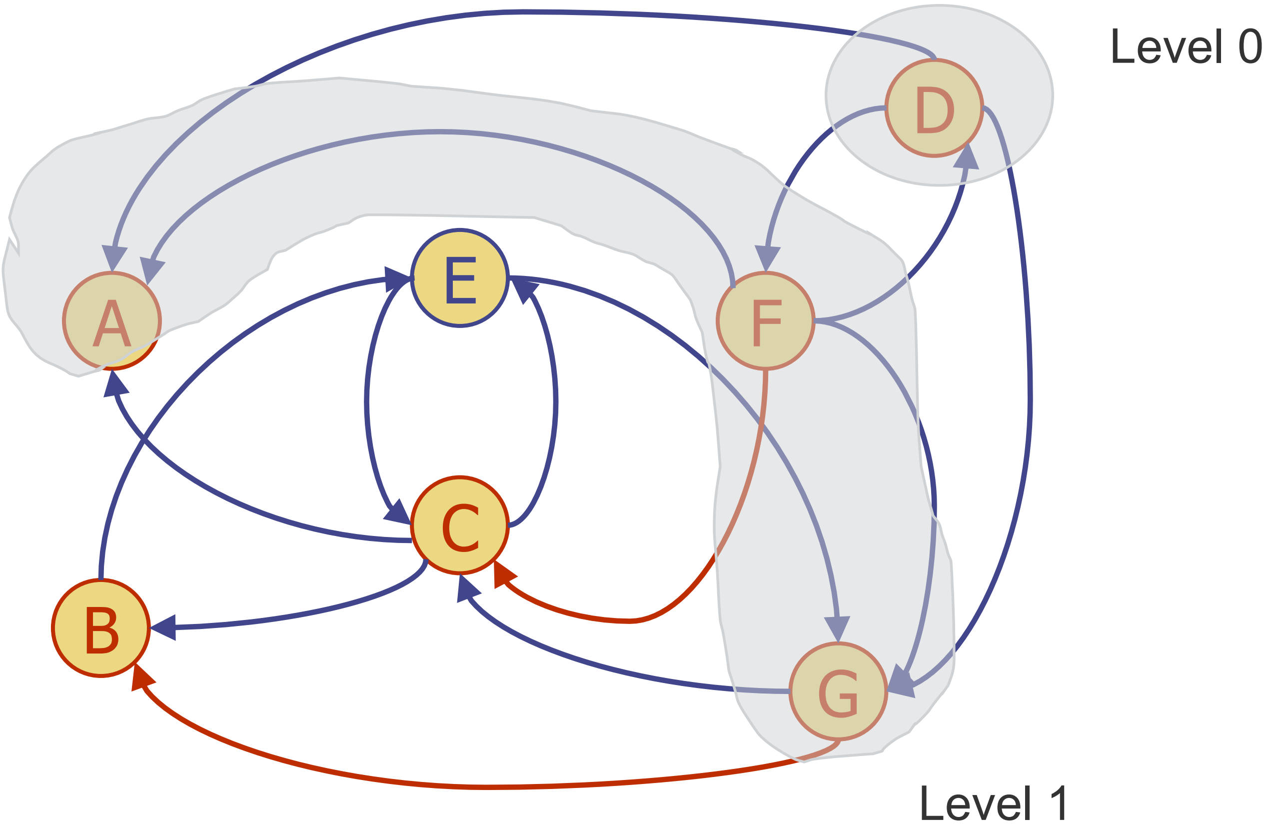
Level 0

visited	discovery edge
D	None
A	(D,A)
F	(D,F)
G	(D,G)

Current level: D
Edges to consider: to A, F, G

- Start from vertex D, which is marked as visited (red)
- Assume that the outgoing edges of a vertex are considered in alphabetical order – e.g. for D: A, F, G

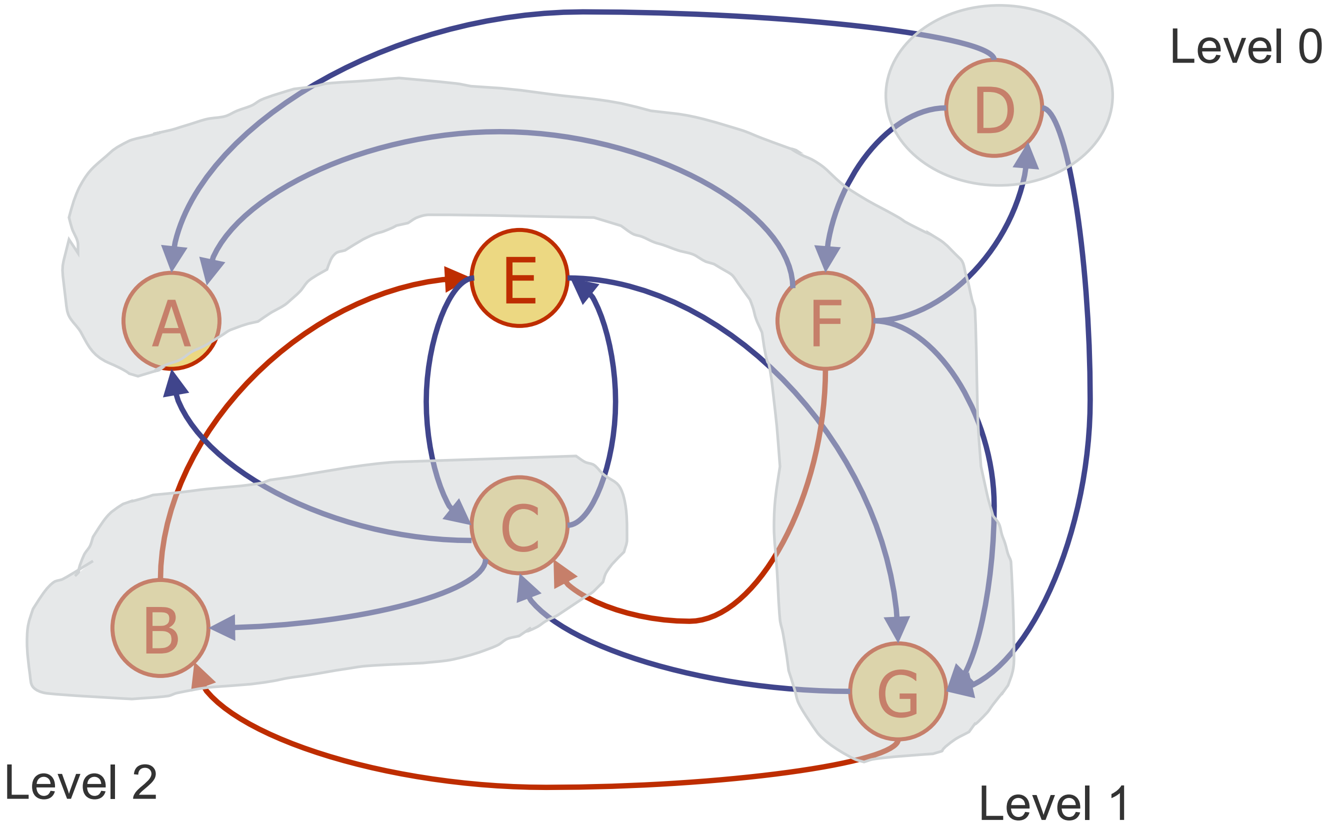
BFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
G	(D,G)
C	(F,C)
B	(G,B)

Current level: A, F, G
Edges to consider: (F, A), (F,C), (F,D), (F,G), (G,B), (G,C)

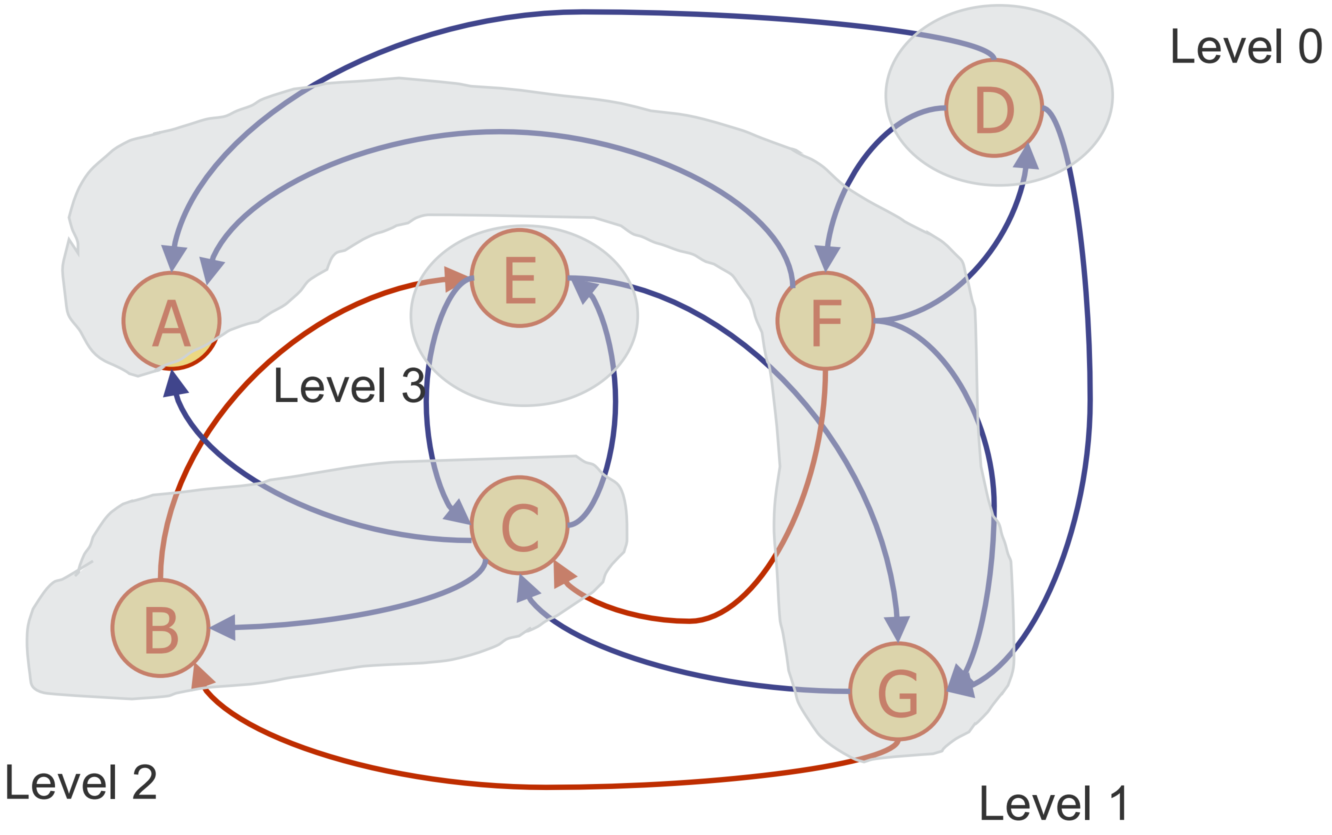
BFS in a Directed Graph - Example



visited	discovery edge
D	None
A	(D,A)
F	(D,F)
G	(D,G)
C	(F,C)
B	(G,B)
E	(B, E)

Current level: B, C
Edges to consider: (B,E), (C, A), (C,B), (C,E)

BFS in a Directed Graph - Example



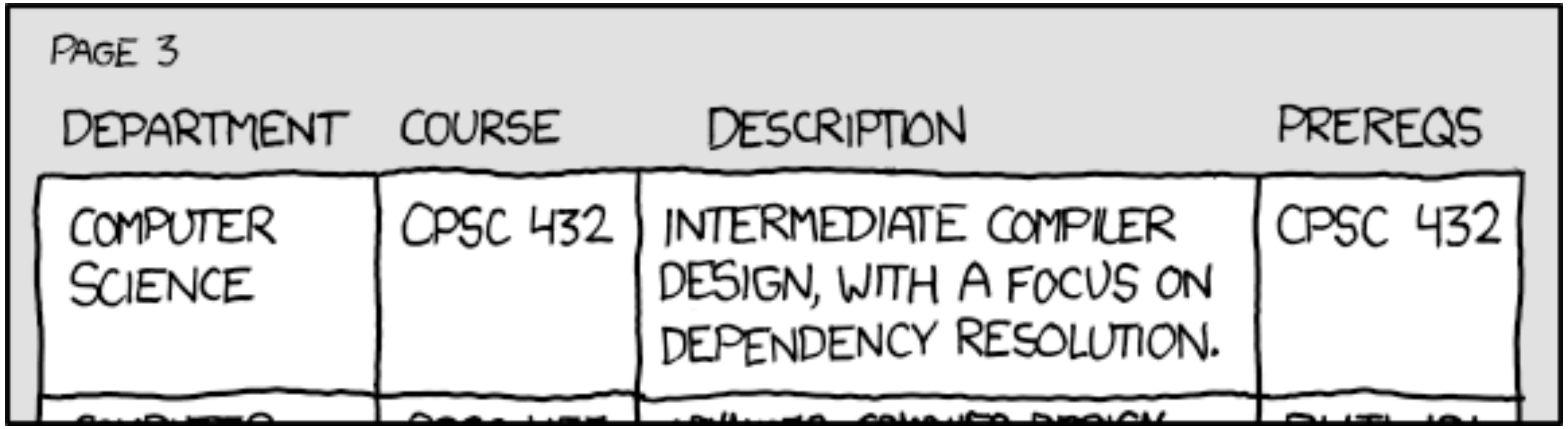
visited	discovery edge
D	None
A	(D,A)
F	(D,F)
G	(D,G)
C	(F,C)
B	(G,B)
E	(B, E)

Current level: E
Edges to consider: (E,C), (E,G)
No new nodes for next level, BFS stop.

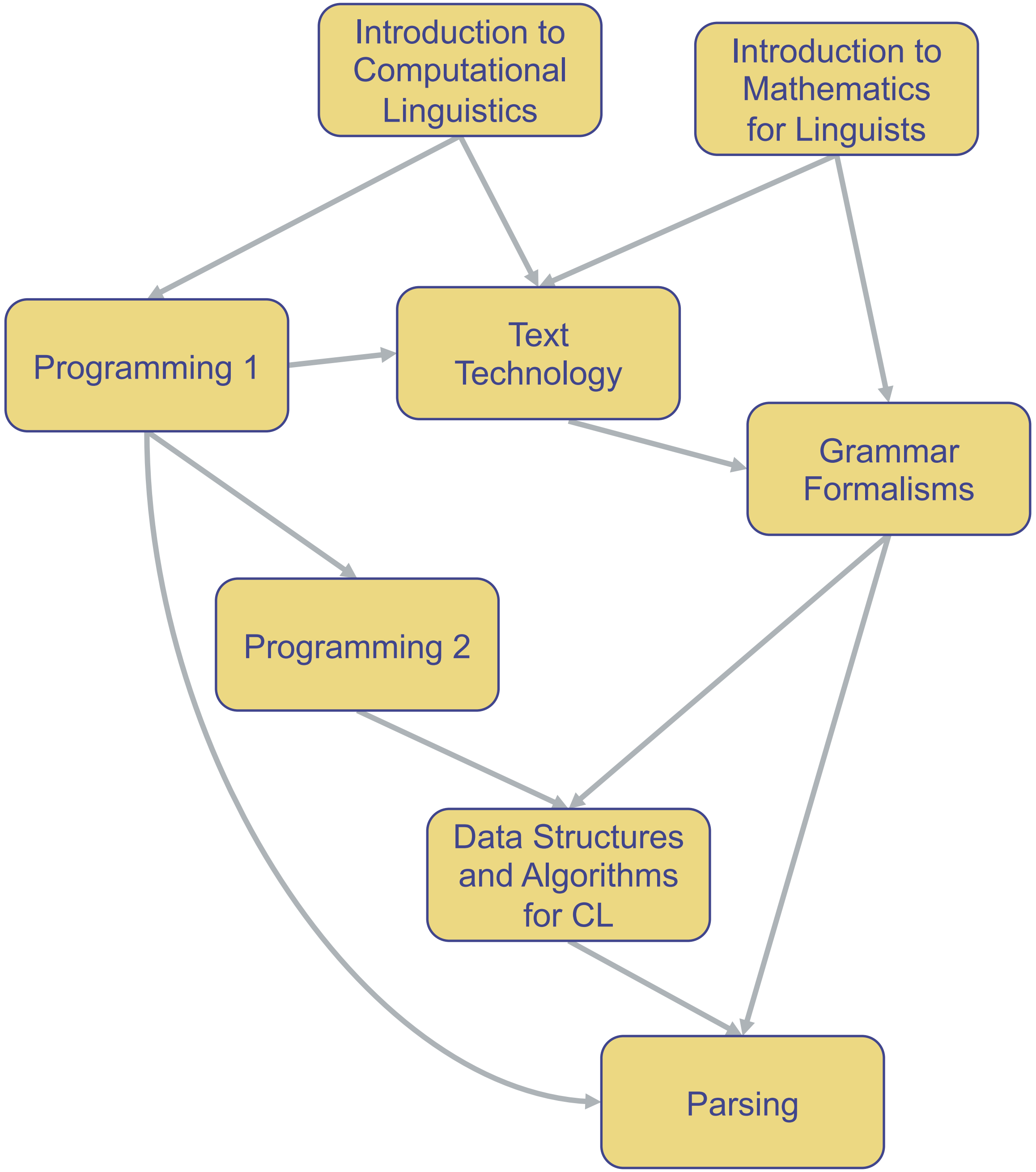
Topological Ordering in Directed Acyclic Graphs (DAGs)

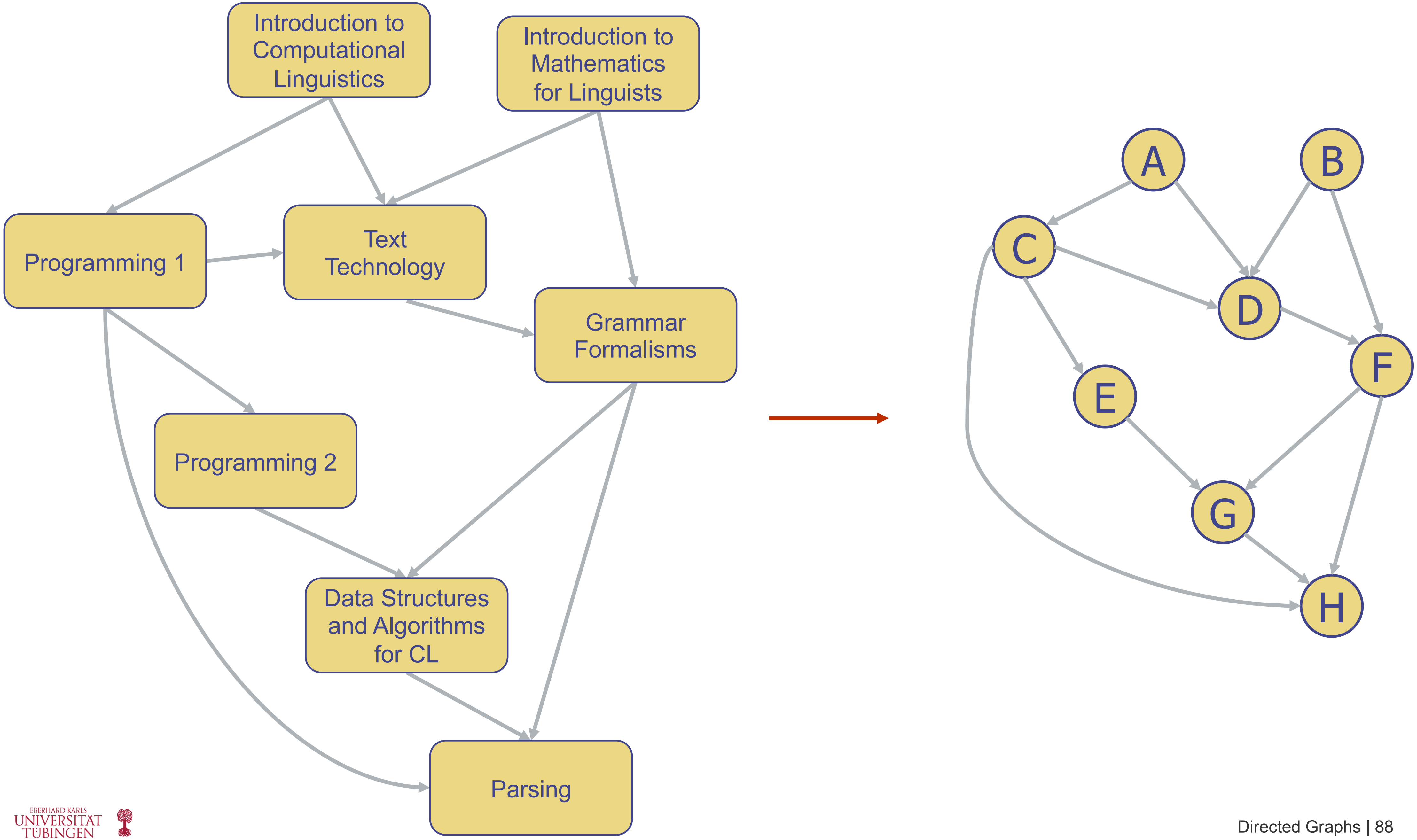
Directed Acyclic Graphs (DAGs)

- directed acyclic graphs are directed graphs without directed cycles
- DAGs are encountered in many practical applications
  - Prerequisites between the courses for a degree program
  - Inheritance between classes of an object-oriented program
  - Scheduling constrains between the tasks of a project



https://www.xkcd.com/754/


Introduction to Computational Linguistics
Introduction to Mathematics for Linguists
Programming 1
Text Technology
Grammar Formalisms
Programming 2
Data Structures and Algorithms for CL
Parsing

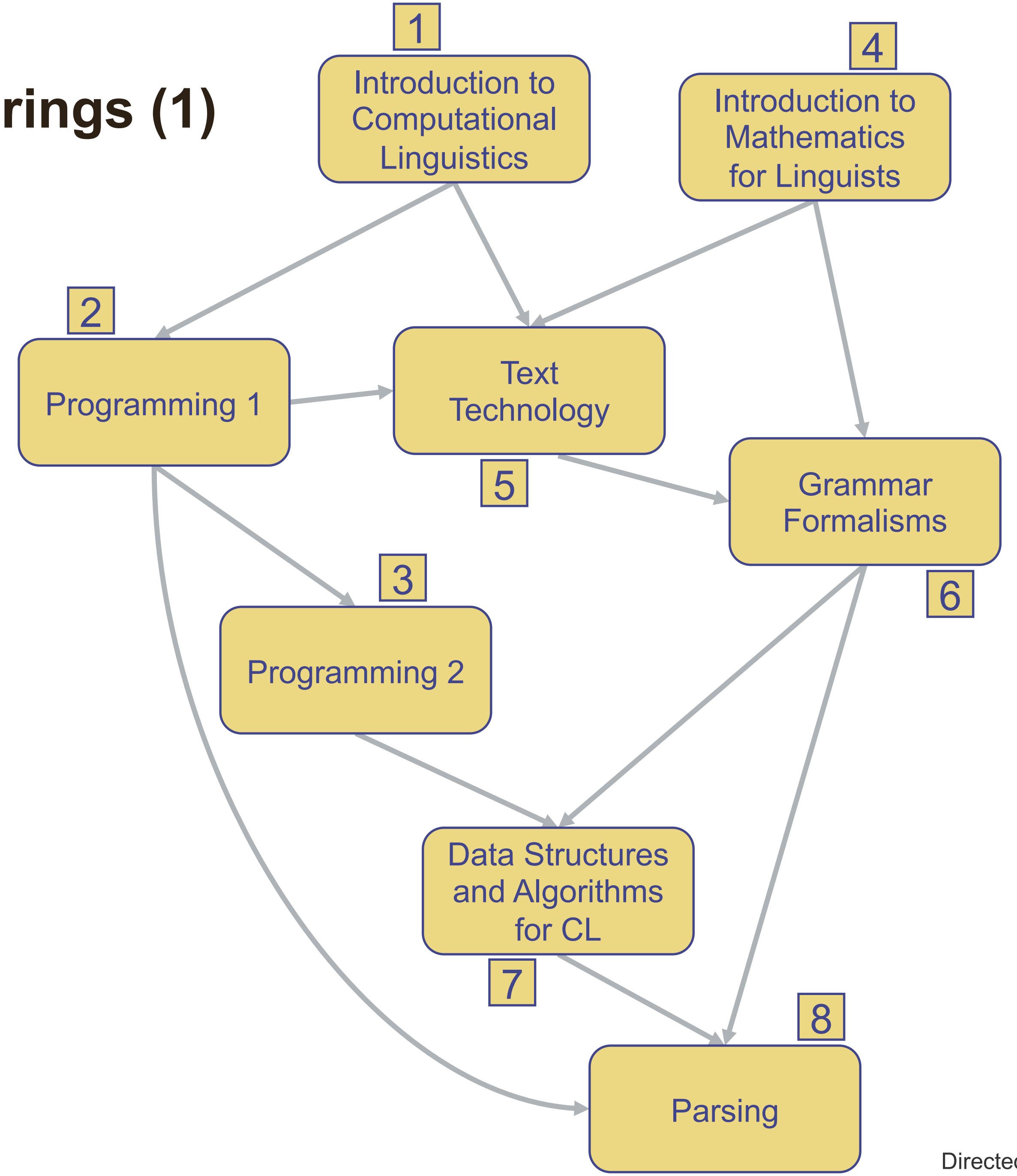


UNIVERSITÄT TÜBINGEN

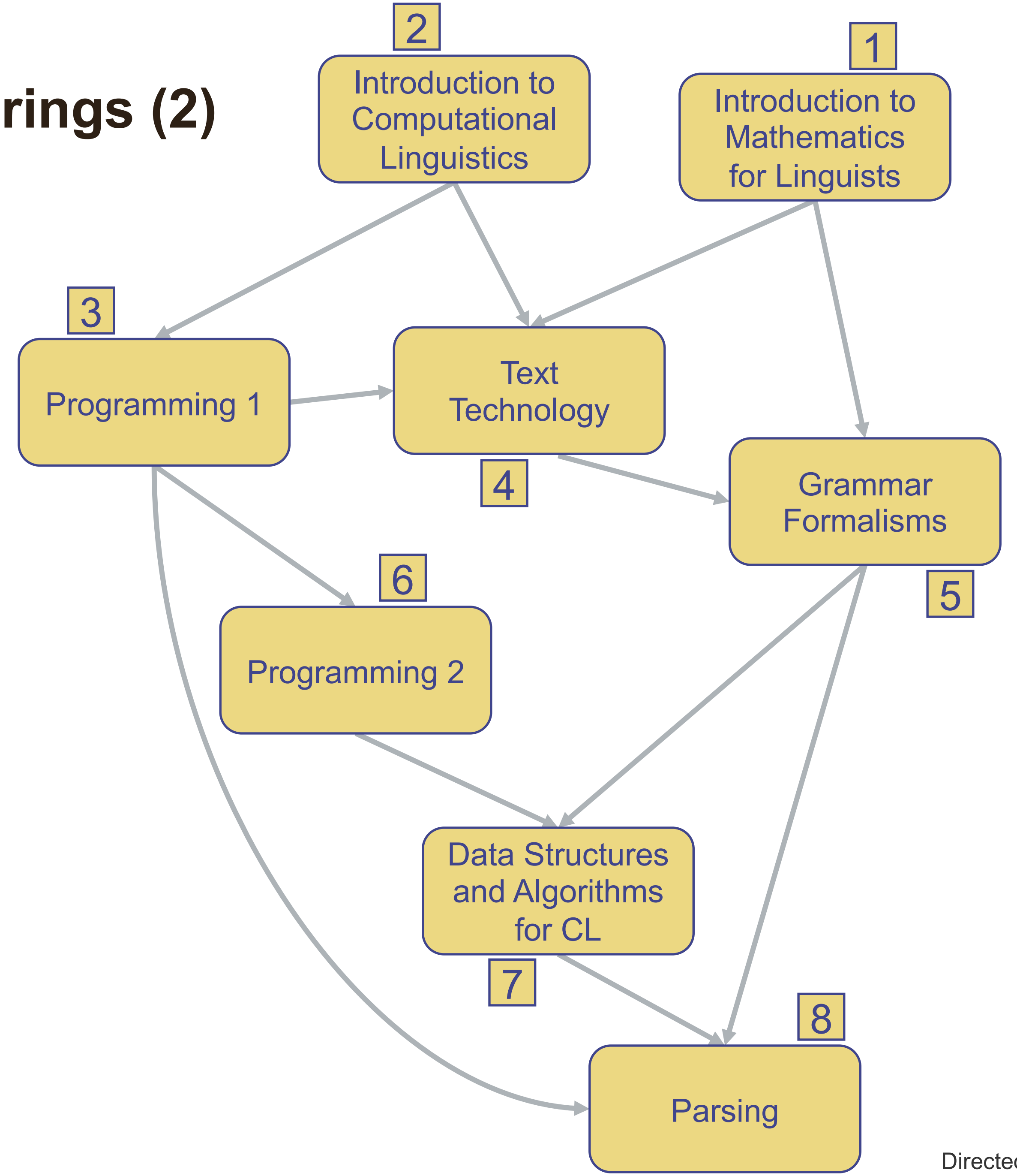
Topological Ordering

- $\vec{G}$ is a directed graph with n vertices
- A topological ordering of $\vec{G}$ is an ordering $v_1, v_2, \dots, v_n$ of the vertices of $\vec{G}$ such that for every edge (v_i, v_j) of $\vec{G}$, it is the case that $i < j$.
- A topological ordering is an ordering such that any directed path in $\vec{G}$ traverses vertices in an increasing order
- A directed graph might have more than one topological orderings

Alterante Topological Orderings (1)



Alterante Topological Orderings (2)



When does a Directed Graph Have a Topological Ordering?

- Proposition. $\vec{G}$ has a topological ordering if and only if it is acyclic.
- Justification.
  - $\Longrightarrow$ Suppose $\vec{G}$ is topologically ordered. Assume that $\vec{G}$ has a cycle made of the edges $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_0})$. But $\vec{G}$ has a topological ordering, meaning that $i_0 < i_1 < \cdots < i_{k-1} < i_0$ - impossible, therefore $\vec{G}$ must be acyclic.

When does a Directed Graph Have a Topological Ordering?

- Proposition. $\vec{G}$ has a topological ordering if and only if it is acyclic.
- Justification.
  - $\Leftarrow$ Suppose $\vec{G}$ is acyclic. A topological ordering can be built using the following algorithm:
    - $\vec{G}$ is acyclic, therefore $\vec{G}$ must have a vertex with no incoming edges, v_1
    - if a vertex like v_1 would not exist, we would eventually encounter a vistied vertex when tracing a path from the start index - would contradict $\vec{G}$ being acyclic
    - thus by removing v_1 and its outgoing edges we obtrain another acyclic graph; this graph has, again, a vertex v_2 with no incoming edges
    - repeat the process of removing the vertex with no incoming edges until $\vec{G}$ is empty
    - $v_1, v_2, \dots, v_n$ form an ordering of the vertices in $\vec{G}$; because of how it was constructed, if (v_i, v_j) is an edge in $\vec{G}$, v_i must be deleted before v_j can be deleted – thus $i < j$ and $v_1, v_2, \dots, v_n$ is a topological ordering

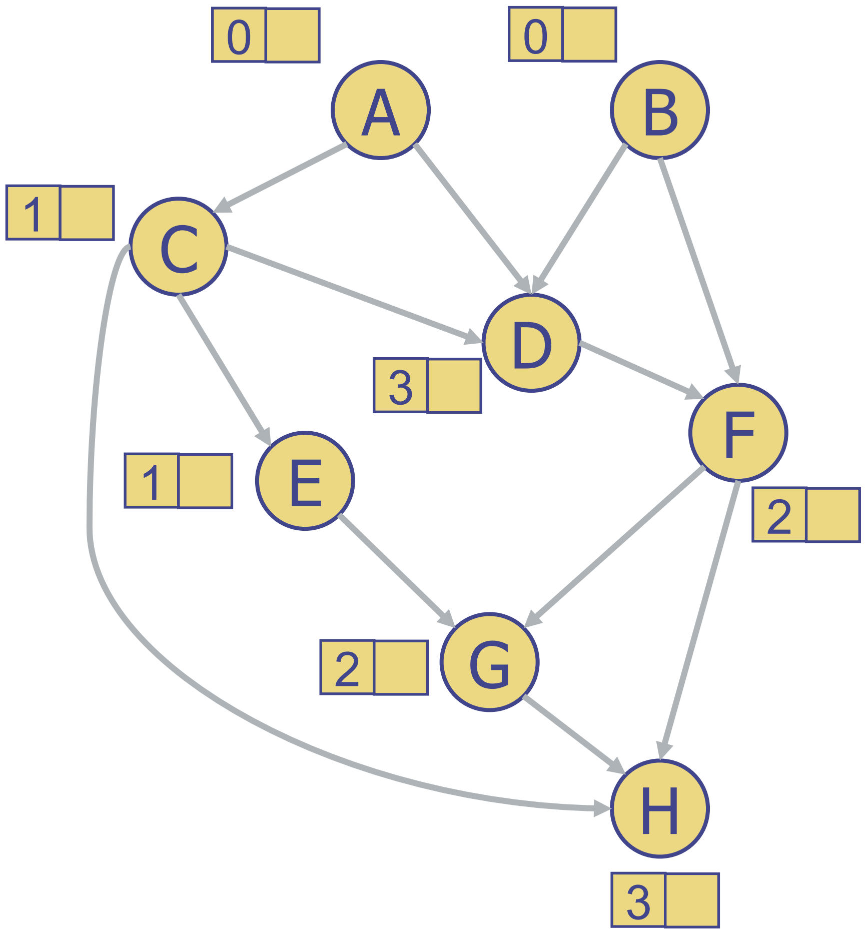
Topological Sorting

```python
def topological_sort(g):
  """Return a list of verticies of directed acyclic graph g in topological order.

  If graph g has a cycle, the result will be incomplete.
  """
  topo = [ ]              # a list of vertices placed in topological order
  ready = [ ]             # list of vertices that have no remaining constraints
  incount = { }           # keep track of in-degree for each vertex
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)    # parameter requests incoming degree
    if incount[u] == 0:                # if u has no incoming edges,
      ready.append(u)                  # it is free of constraints
  while len(ready) > 0:
    u = ready.pop( )                   # u is free of constraints
    topo.append(u)                     # add u to the topological order
    for e in g.incident_edges(u):      # consider all outgoing neighbors of u
      v = e.opposite(u)
      incount[v] −= 1                  # v has one less constraint without u
      if incount[v] == 0:
        ready.append(v)
  return topo
```

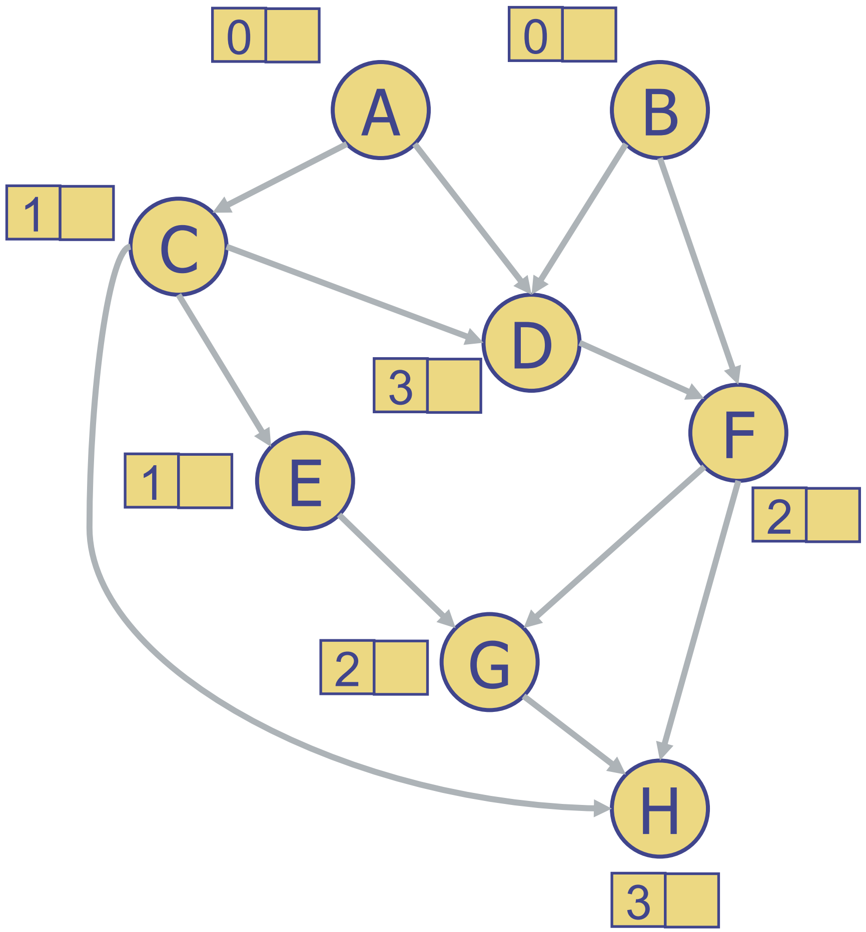
- topological sorting is an algorithm for computing a topological ordering of a directed graph
- `incount` is a `dict`, maps
  - vertex u to
  - number of incoming edges to u (excluding those from vertices that have been added to the topological order)
- also tests if $\vec{G}$ is acyclic: if the algorithm terminates without ordering all the vertices, then the subgraph of vertices that have not been ordered must contain a cycle

Topological Sorting - Example



- in the left box, the current incount of each of the vertices in the graph
- in the right box, the index of the vertex in the topological ordering

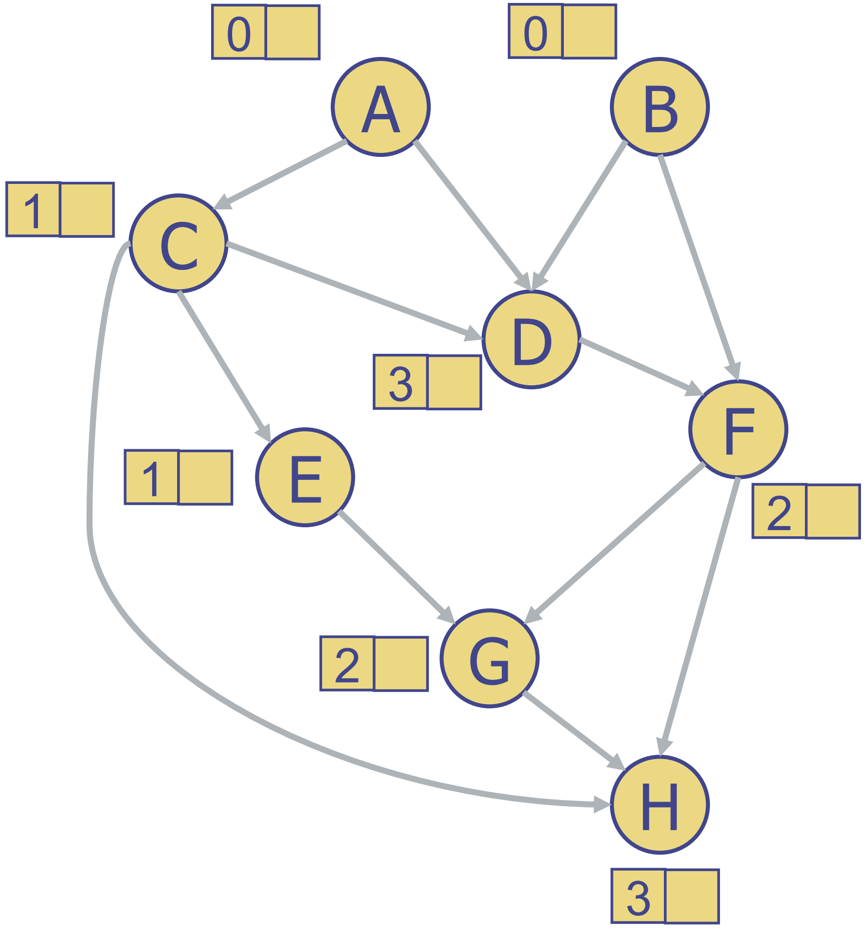
Topological Sorting - Example



topo	ready
	B
	A

- len(ready) > 0 == True

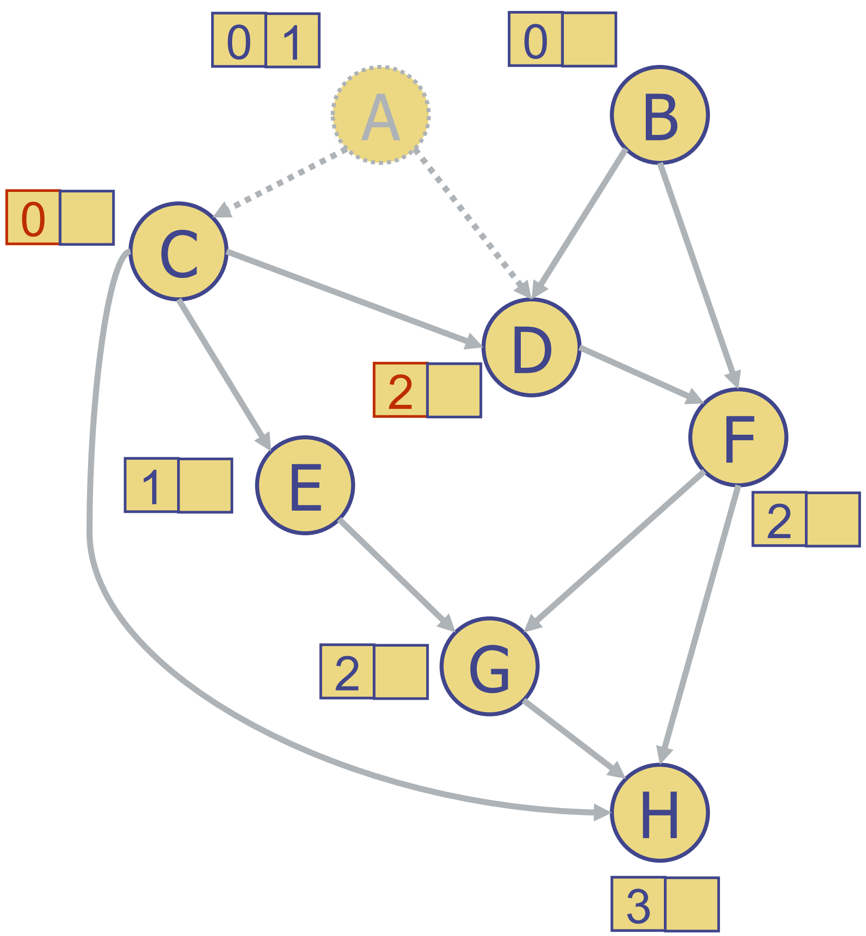
Topological Sorting - Example



topo	ready
A	B

- pop A from ready, append it to topo
- decrease the incount of all neighbours of A (on outgoing edges): C, D

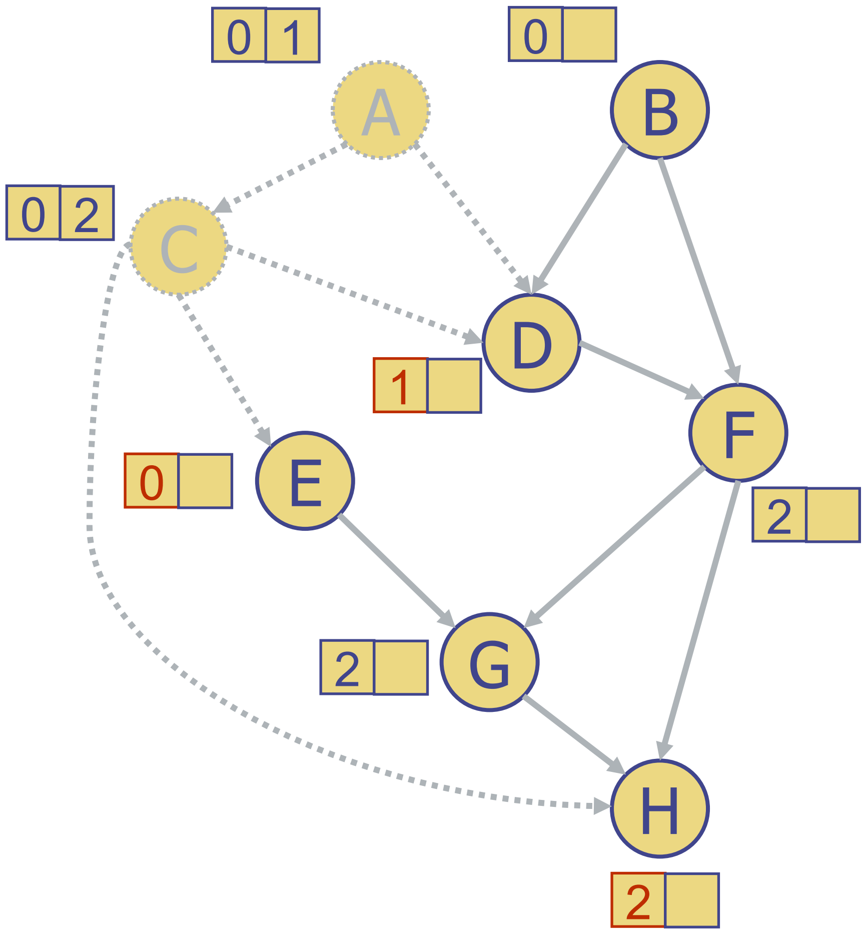
Topological Sorting - Example



topo	ready
A	B
	C

- if after the decrease any of the vertices have an incount of 0, add it to ready
- pop C, add it to topo

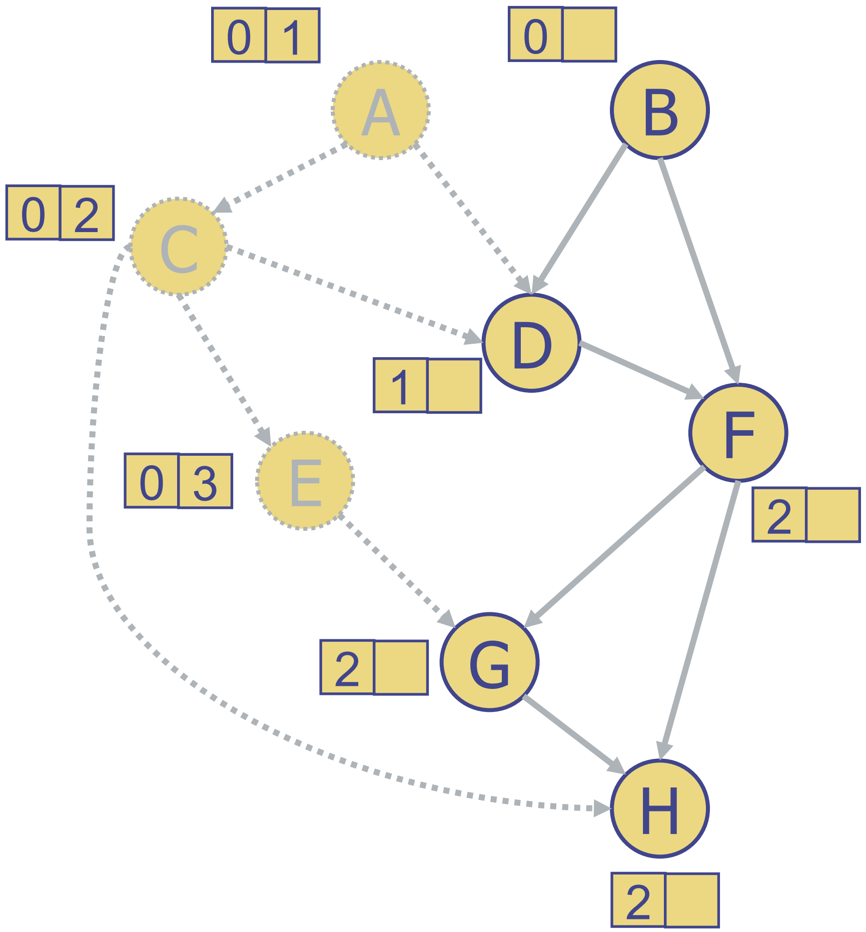
Topological Sorting - Example



topo	ready
A	B
C	E

- decrease the incount of D, E and H
- add E to ready, since its incount is 0

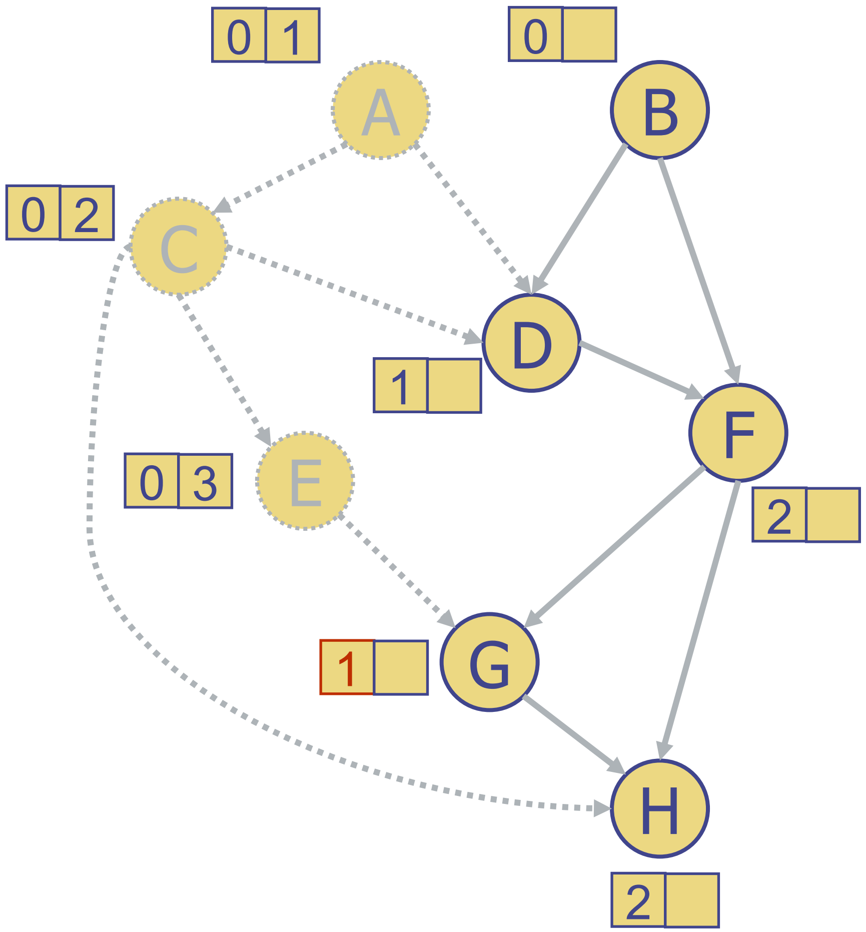
Topological Sorting - Example



topo	ready
A	B
C	
E	

- pop E from ready, add it to topo

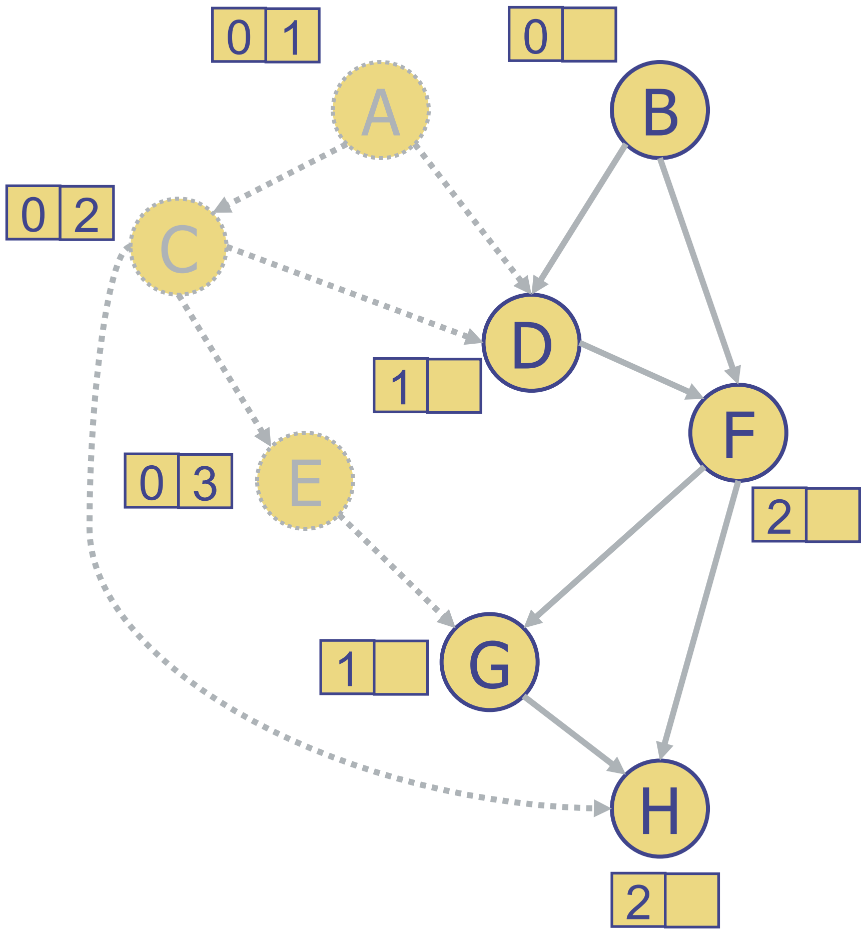
Topological Sorting - Example



topo	ready
A	B
C	
E	

- decrease the incount of G

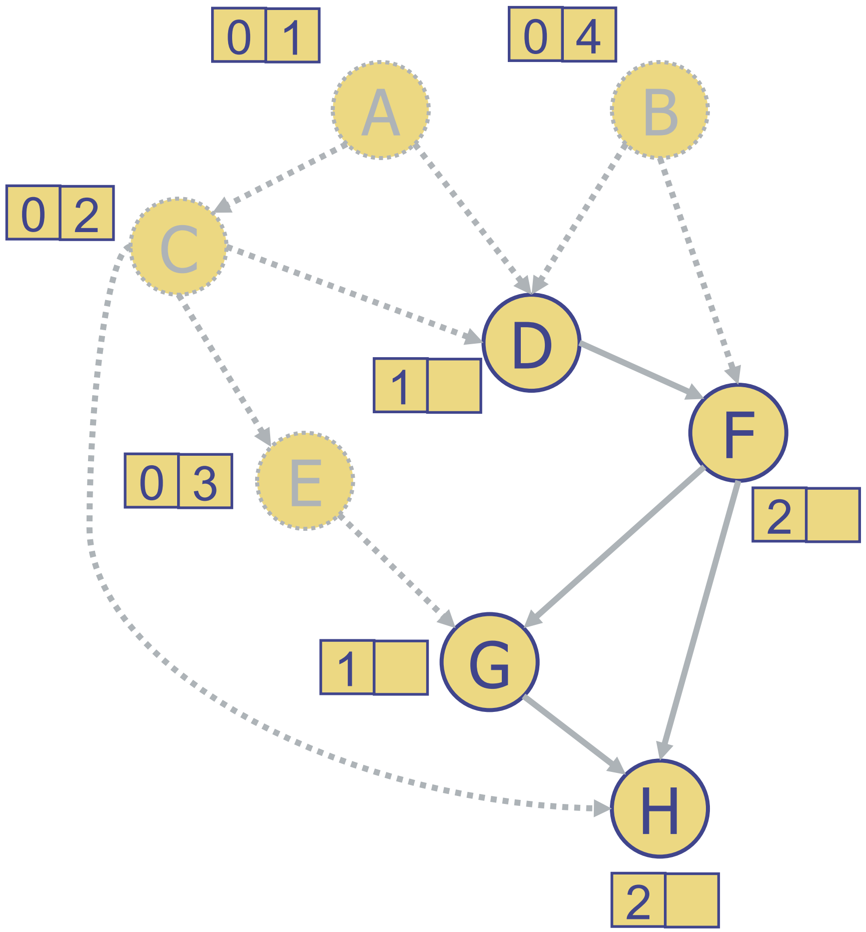
Topological Sorting - Example



topo	ready
A	B
C	
E	

- pop B from ready, add it to topo

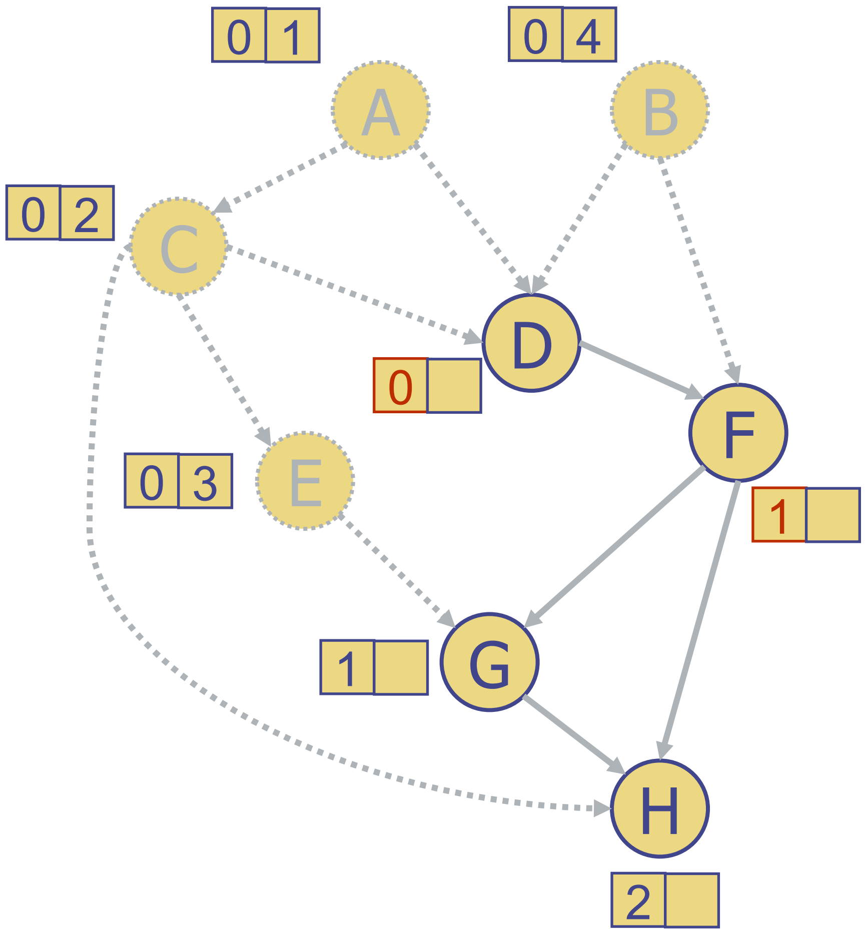
Topological Sorting - Example



topo	ready
A	
C	
E	
B	

- pop B from ready, add it to topo

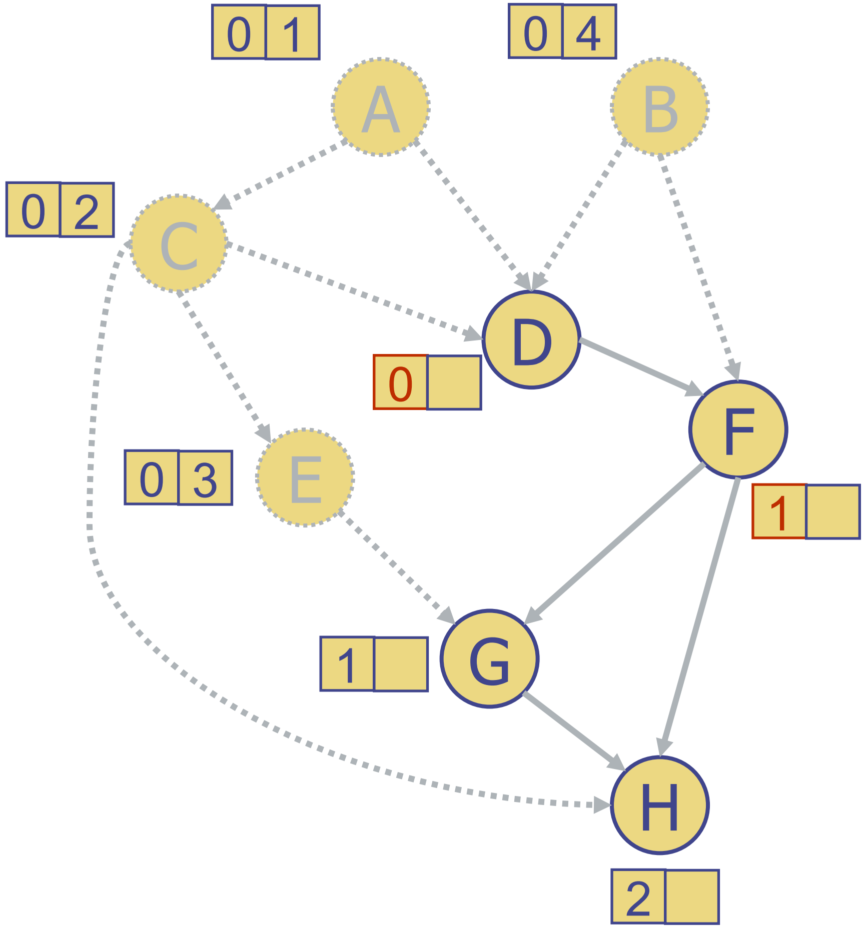
Topological Sorting - Example



topo	ready
A	
C	
E	
B	

- decrease the incount of D and F

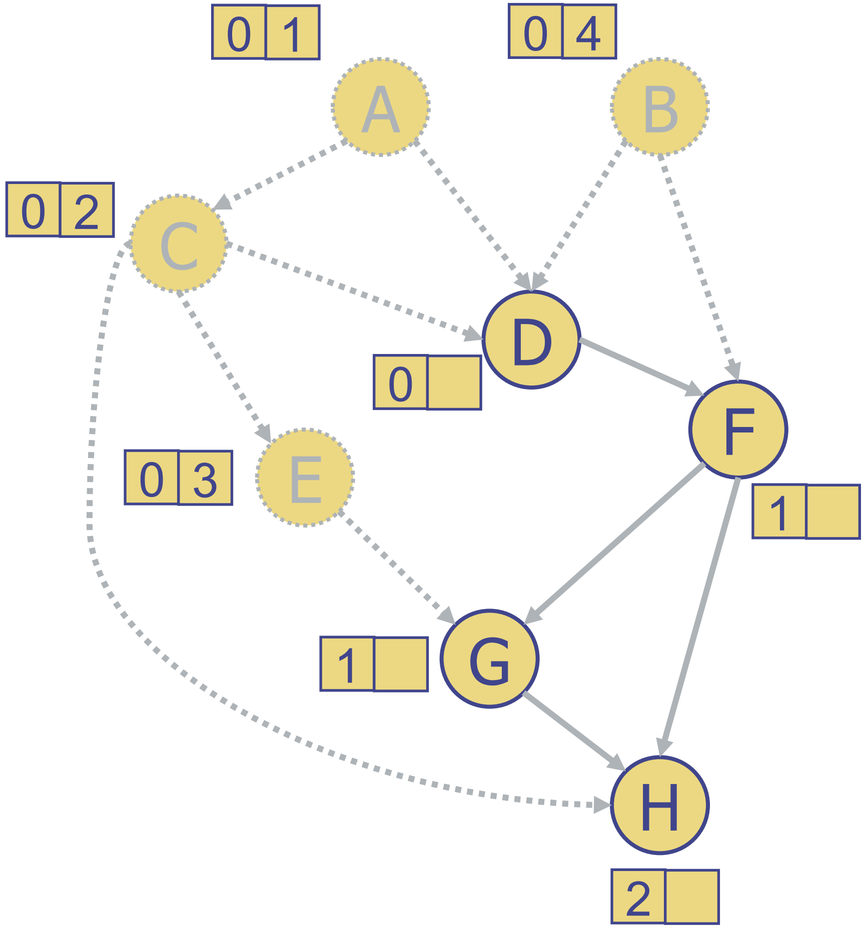
Topological Sorting - Example



topo	ready
A	D
C	
E	
B	

- add D to ready, since its incount is 0

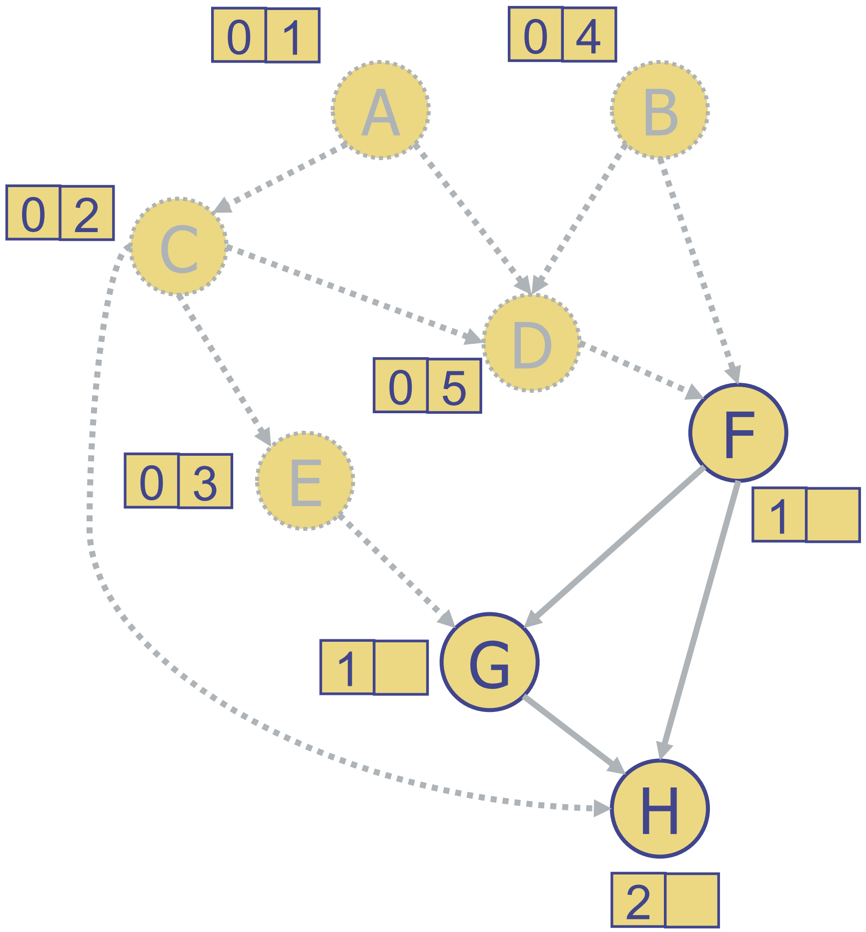
Topological Sorting - Example



topo	ready
A	D
C	
E	
B	

- pop D from ready, add it to topo

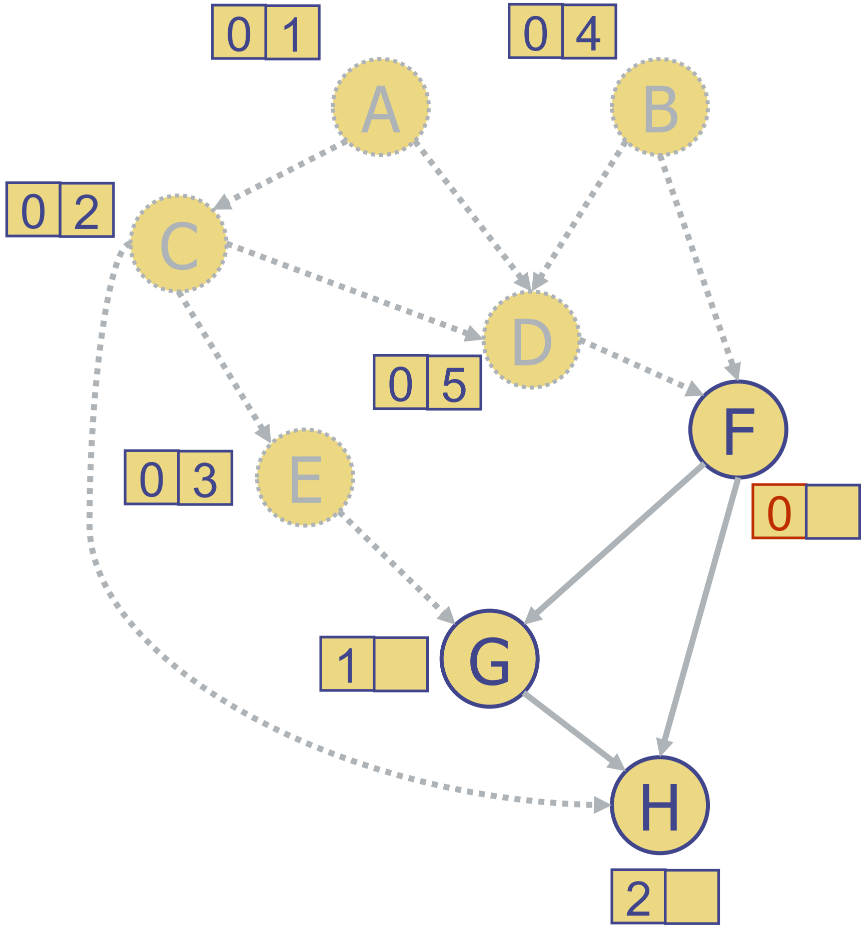
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	

- decrement the incount of F

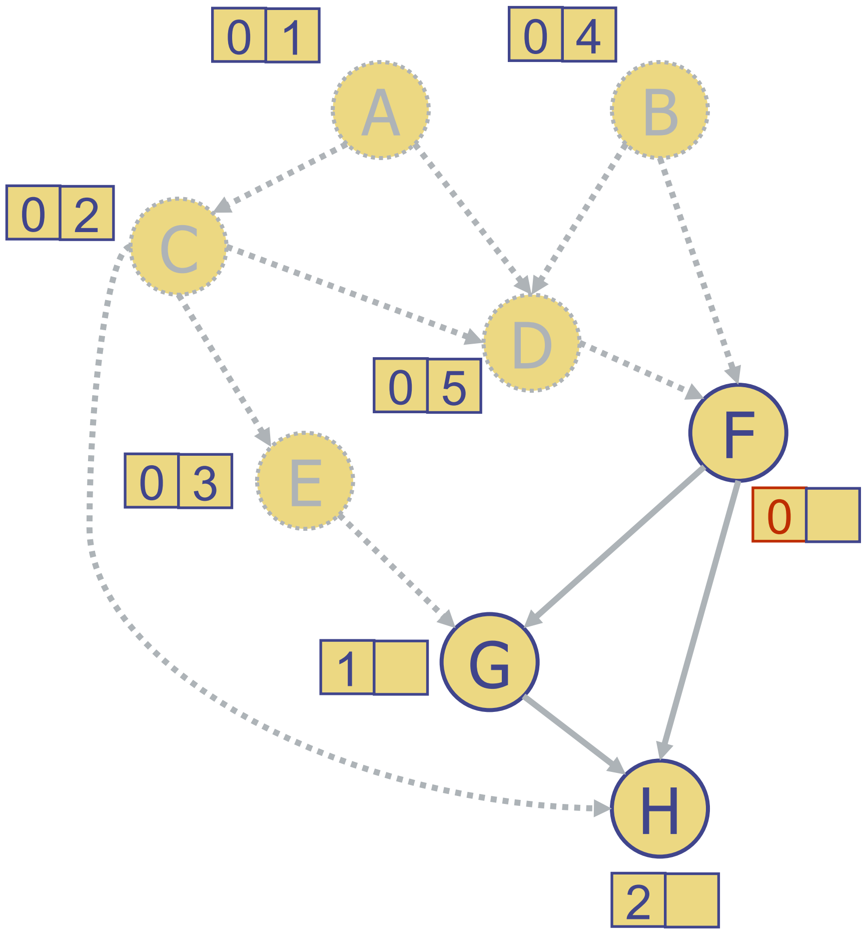
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	

- decrement the incount of F

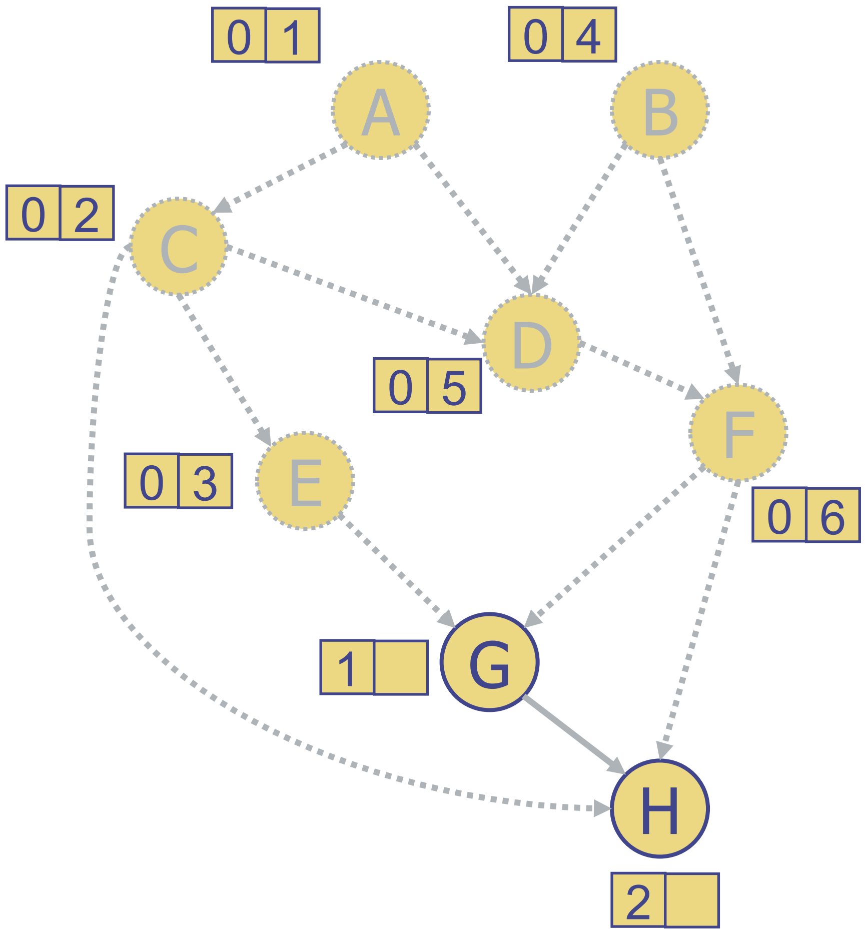
Topological Sorting - Example



topo	ready
A	F
C	
E	
B	
D	

- add F to ready, its incount is 0

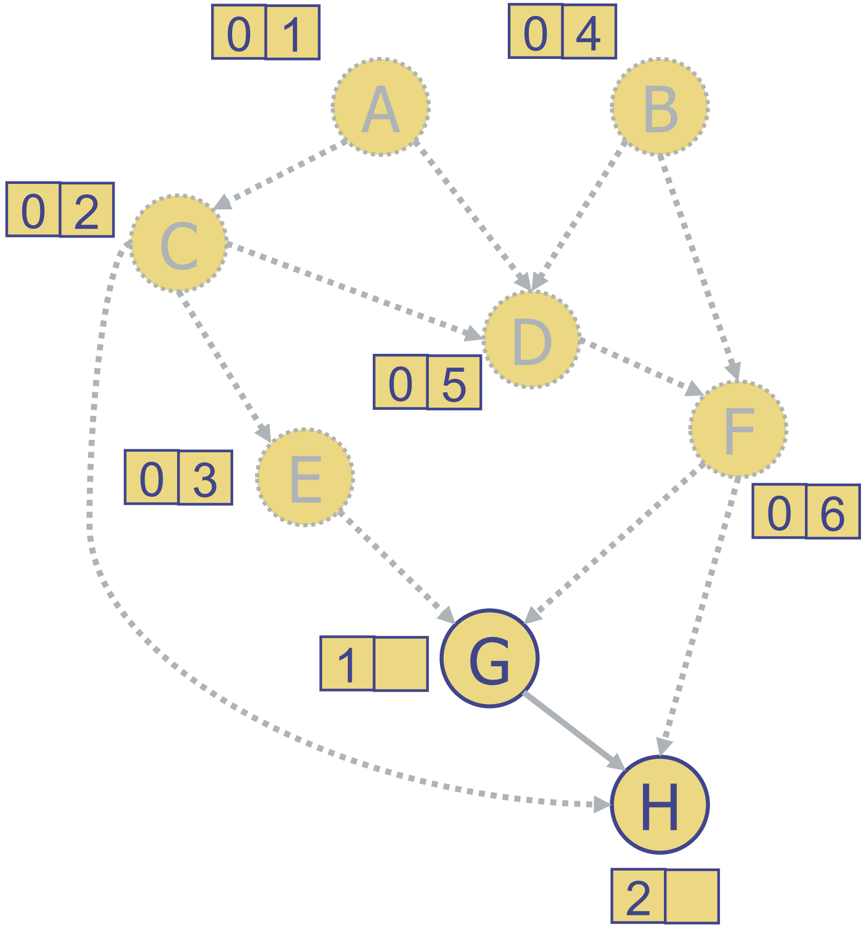
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	

- pop F from ready, add to topo

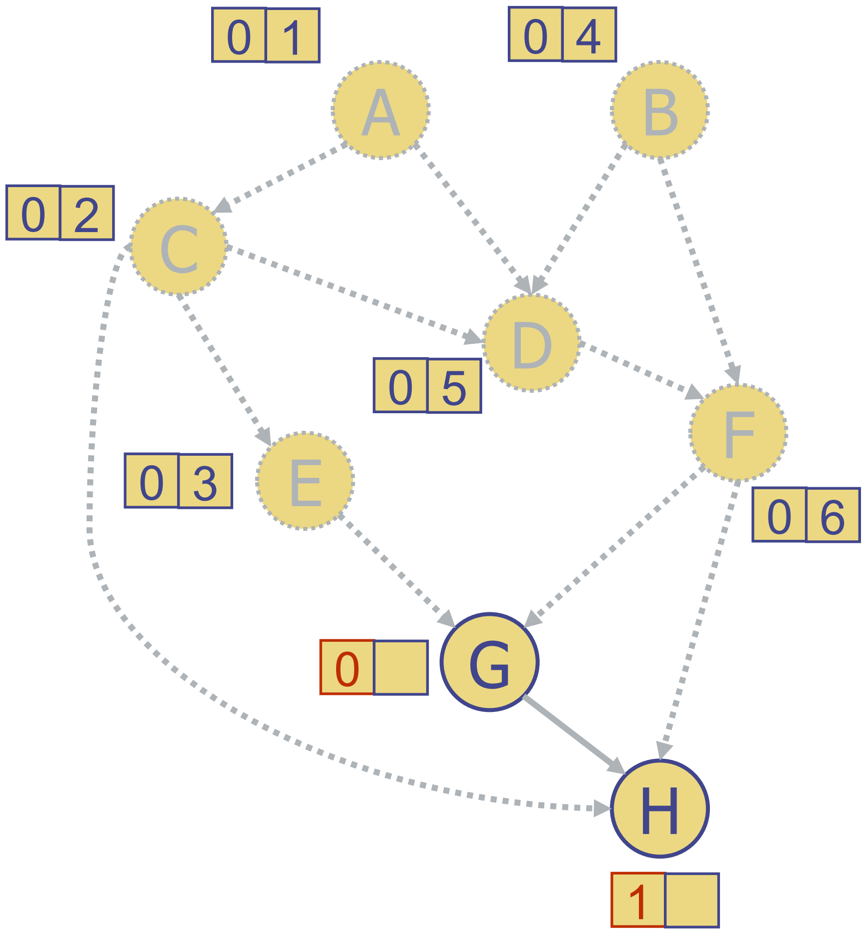
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	

- decrement the incounts of G and H

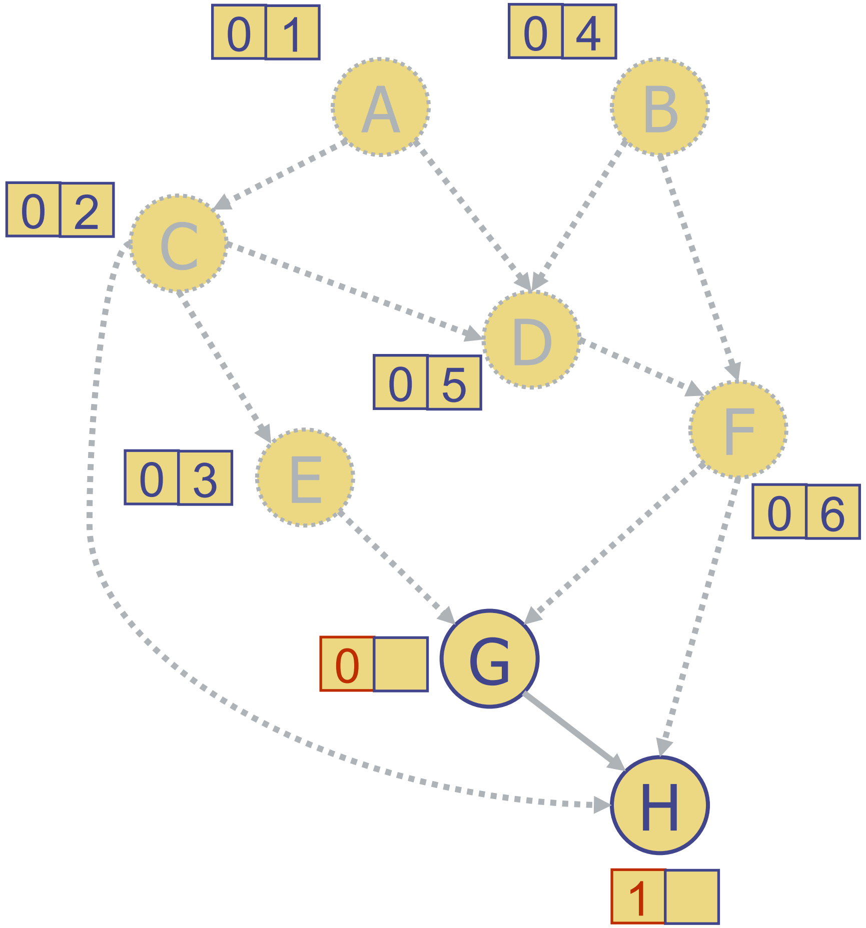
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	

- decrement the incounts of G and H

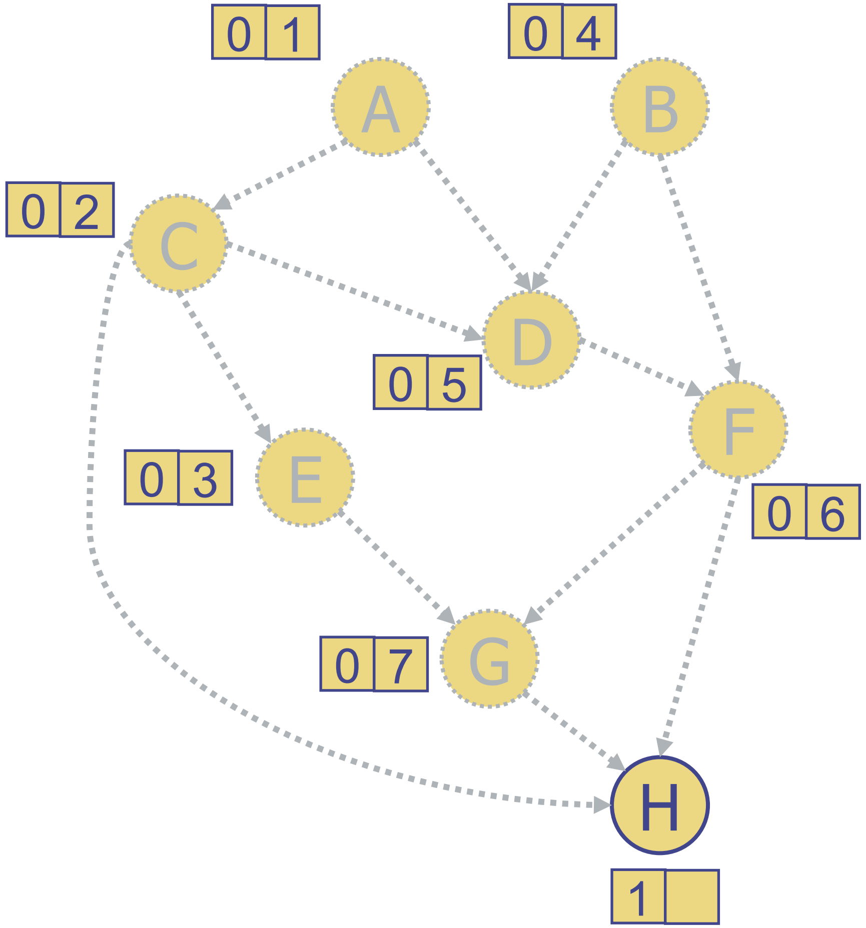
Topological Sorting - Example



topo	ready
A	G
C	
E	
B	
D	
F	

- add G to ready, its incount is 0

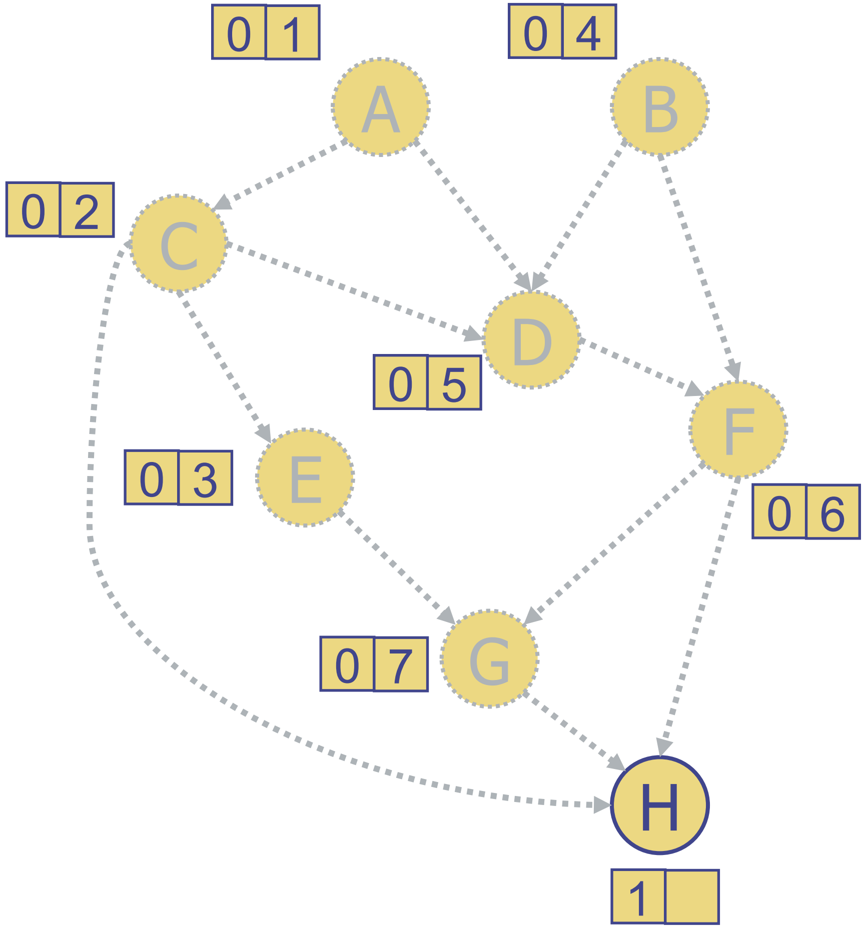
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	
G	

- pop G from ready, add to topo

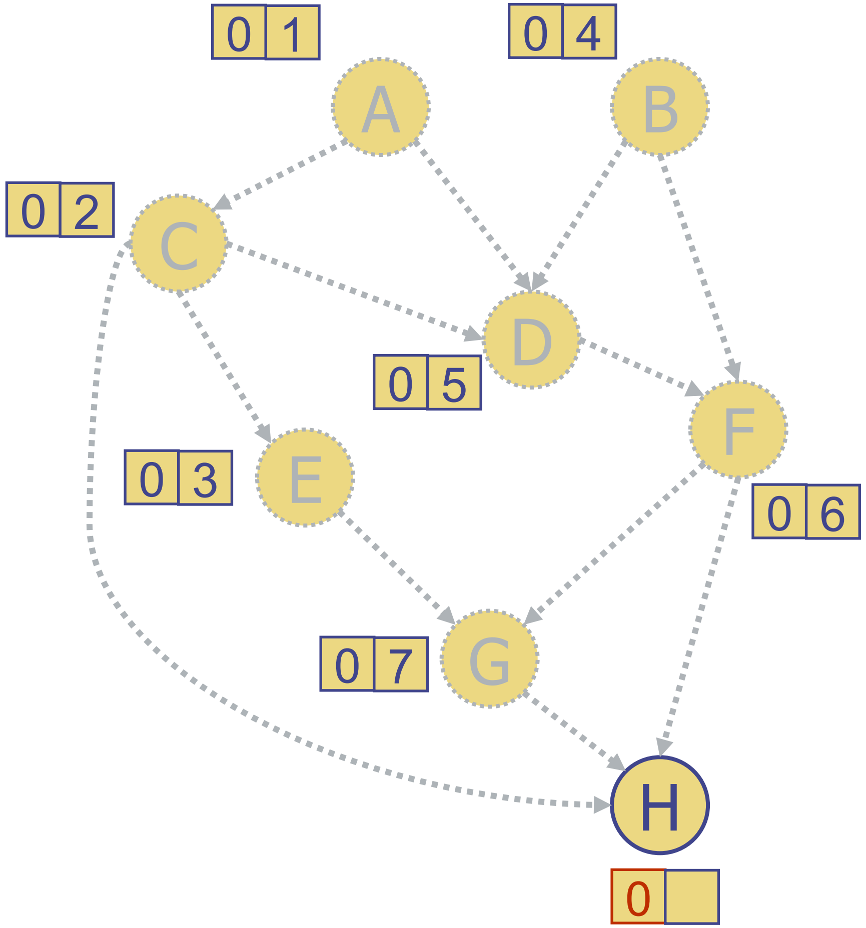
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	
G	

- decrement the incount of H

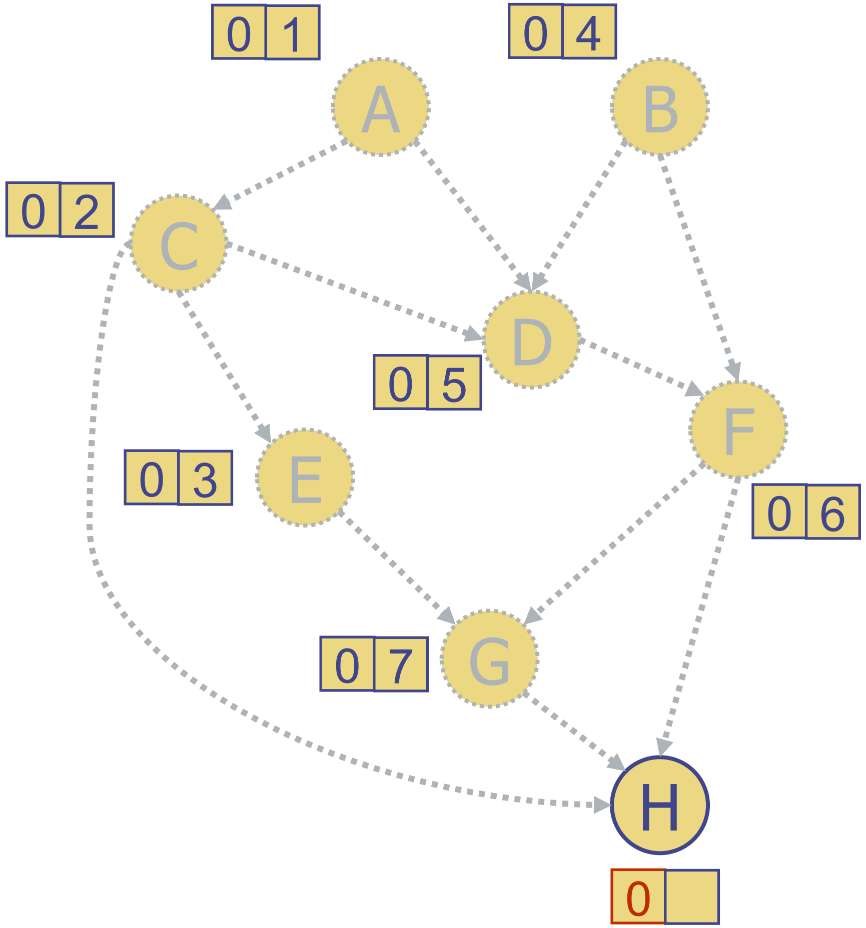
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	
G	

- decrement the incount of H

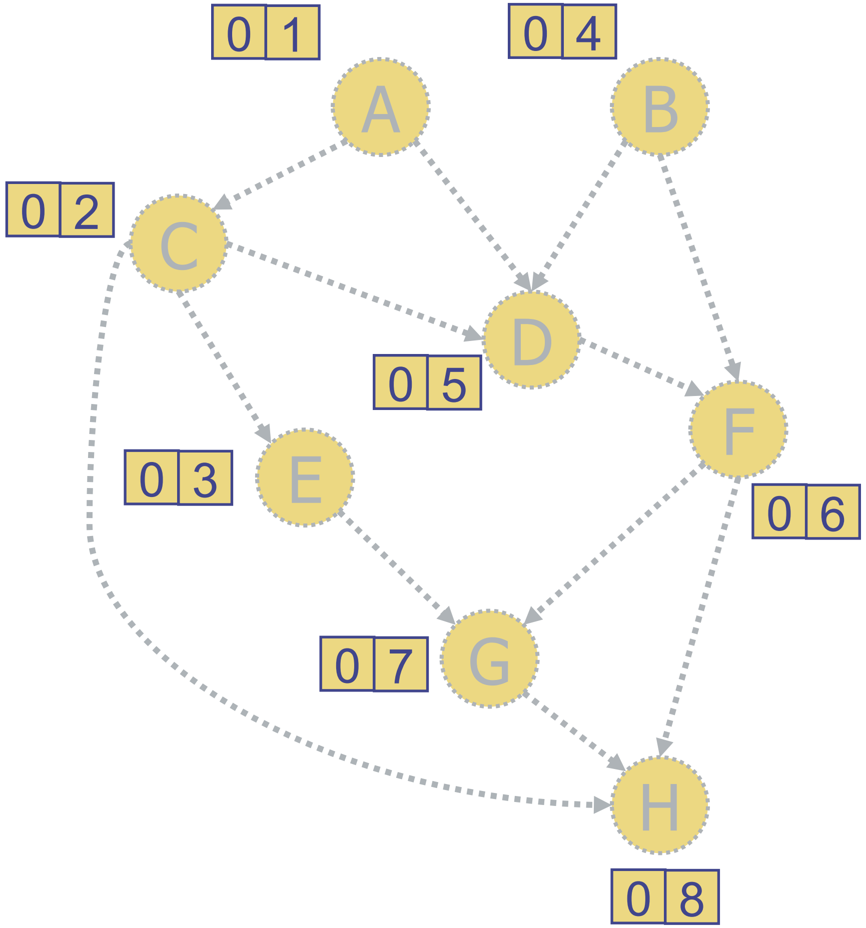
Topological Sorting - Example



topo	ready
A	H
C	
E	
B	
D	
F	
G	

- add H to ready, its incount is 0

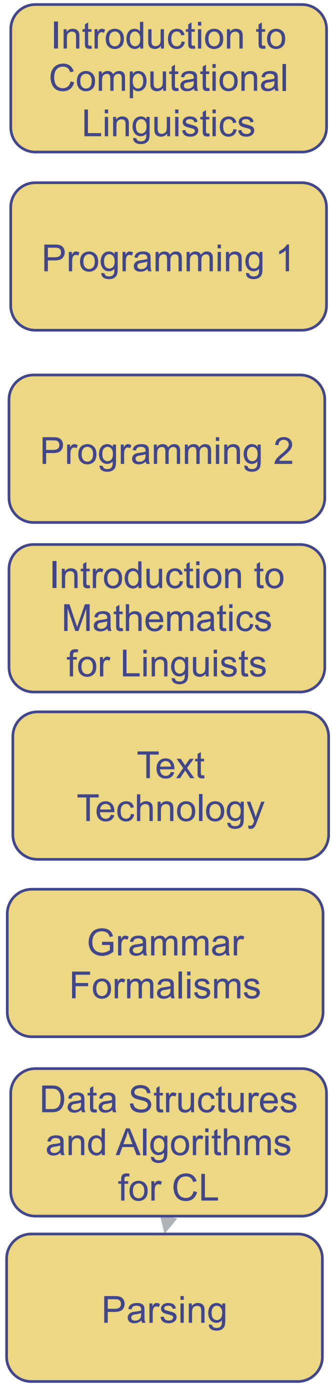
Topological Sorting - Example



topo	ready
A	
C	
E	
B	
D	
F	
G	
H	

- pop H from ready, add to topo
- H has no outgoing nodes
- ready is empty – stop.
- the topological ordering of the vertices is obtained in topo

topo
A
C
E
B
D
F
G
H



Topological Sorting - Performance

```
def topological_sort(g):
  """Return a list of verticies of directed acyclic graph g in topological order.

  If graph g has a cycle, the result will be incomplete.
  """
  topo = [ ]              # a list of vertices placed in topological order
  ready = [ ]             # list of vertices that have no remaining constraints
  incount = { }           # keep track of in-degree for each vertex
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)     # parameter requests incoming degree
    if incount[u] == 0:                 # if u has no incoming edges,
      ready.append(u)                   # it is free of constraints
  while len(ready) > 0:
    u = ready.pop( )                    # u is free of constraints
    topo.append(u)                      # add u to the topological order
    for e in g.incident_edges(u):       # consider all outgoing neighbors of u
      v = e.opposite(u)
      incount[v] −= 1                   # v has one less constraint without u
      if incount[v] == 0:
        ready.append(v)
  return topo
```

- topological sorting runs in $O(n+m)$ time, using $O(n)$ auxiliary space
- it either computes a topological ordering of $\vec{G}$ or fails to include some vertices – meaning that $\vec{G}$ has a directed cycle

Thank you.