15-0: **Graphs**

- A graph consists of:
  - A set of **nodes** or **vertices** (terms are interchangeable)
  - A set of **edges** or **arcs** (terms are interchangeable)
- Edges in graph can be either directed or undirected

15-1: **Graphs & Edges**

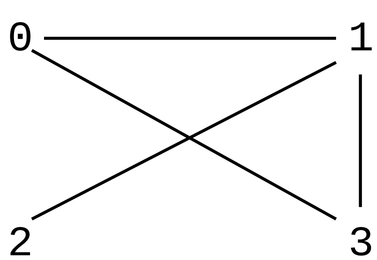
- Edges can be labeled or unlabeled
  - Edge labels are typically the *cost* associated with an edge
  - e.g., Nodes are cities, edges are roads between cities, edge label is the length of road

15-2: **Graph Representations**

- Adjacency Matrix
- Represent a graph with a two-dimensional array G
  - $G[i][j] = 1$ if there is an edge from node i to node j
  - $G[i][j] = 0$ if there is no edge from node i to node j
- If graph is undirected, matrix is symmetric
- Can represent edges labeled with a cost as well:
  - $G[i][j] =$ cost of link between i and j
  - If there is no direct link, $G[i][j] = \infty$

15-3: **Adjacency Matrix**

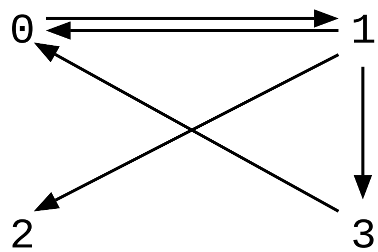
- Examples:



	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	1	0	0
3	1	1	0	0

15-4: **Adjacency Matrix**

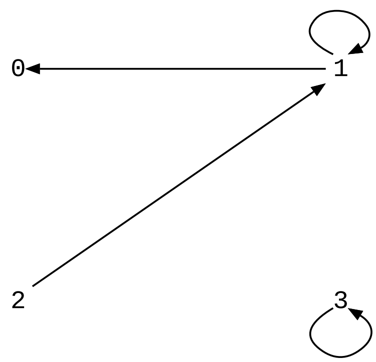
- Examples:



	0	1	2	3
0	0	1	0	0
1	1	0	1	1
2	0	0	0	0
3	1	0	0	0

15-5: **Adjacency Matrix**

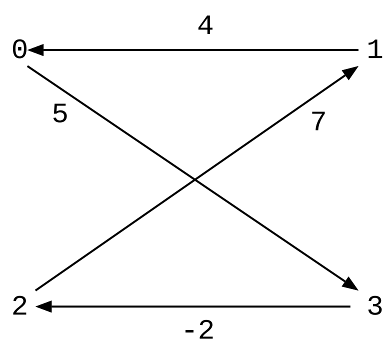
- Examples:



	0	1	2	3
0	0	0	0	0
1	1	1	0	0
2	0	1	0	0
3	0	0	0	1

15-6: **Adjacency Matrix**

- Examples:



	0	1	2	3
0	∞	∞	∞	5
1	4	∞	∞	∞
2	∞	7	∞	∞
3	∞	∞	-2	∞

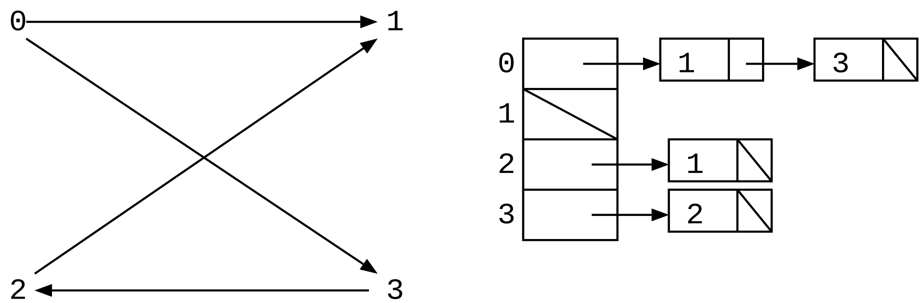
15-7: **Graph Representations**

- Adjacency List
- Maintain a linked-list of the neighbors of every vertex.
  - n vertices

- Array of n lists, one per vertex
- Each list i contains a list of all vertices adjacent to i.

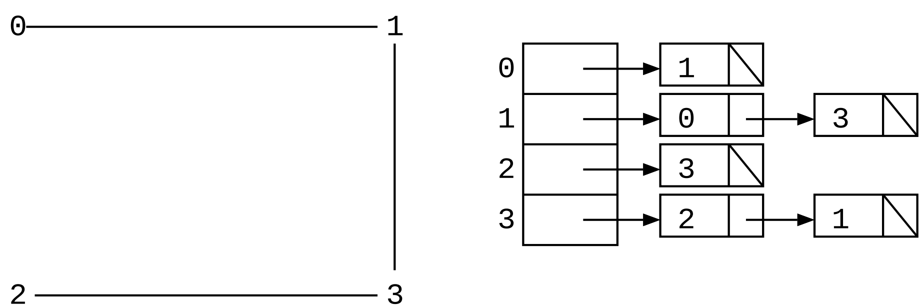
15-8: **Adjacency List**

- Examples:



15-9: **Adjacency List**

- Examples:



- Note – lists are not always sorted

15-10: **Sparse vs. Dense**

- Sparse graph – relatively few edges
- Dense graph – lots of edges
- Complete graph – contains all possible edges
    - These terms are fuzzy. “Sparse” in one context may or may not be “sparse” in a different context

15-11: **Breadth-First Search**

- Method for searching a graph
- Specify a source node in the grap
- Find all nodes reachable from that node
    - First find all nodes 1 unit away
    - Next find all nodes 2 units away

- ... etc

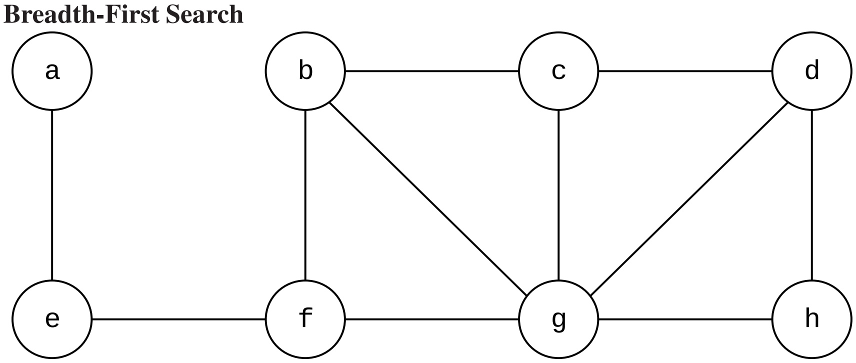
15-12: **Breadth-First Search**

- Auxiliary Data Structures
  - “color” for each vertex – white, black, grey
    - Used to make sure we don’t visit vertices more than once
  - Parent of each vertex (Path to source node)
  - Distance of each vertex from source

15-13: **Breadth-First Search**

```
BFS(G, s)
  for each vertex u in V[G] do
    color[u] ← WHITE
    d[u] ← ∞
    π[u] ← nil
  color[s] ← GRAY
  d[s] ← 0
  Q ← {s}
  while Q not empty do
    u ← Q.dequeue
    for each v adj. to u
      if color[v] = WHITE
        color[v] ← GRAY
        d[v] ← d[u] + 1
        π[v] ← u
        Q.enqueue(v)
    color[u] ← BLACK
```

15-14: **Breadth-First Search**

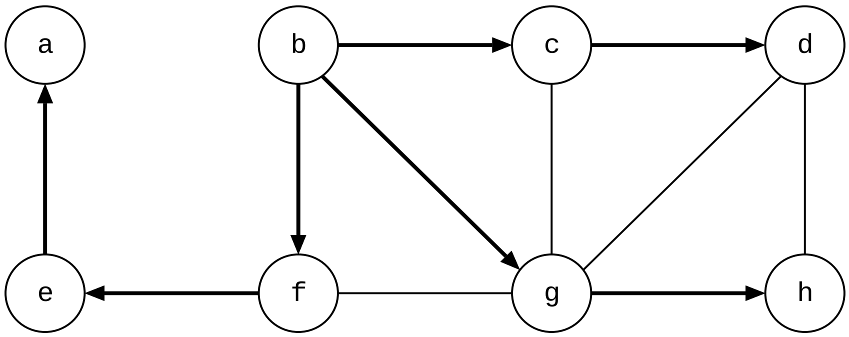


`Q: b`

15-15: **Breadth-First Search**

- BFS computes the shortest path from the start vertex to every other vertex
- We can run BFS on a directed or undirected tree
- Defines a “BFS Tree”
  - Parent pointers $p[v]$
  - BFS Tree is directed

15-16: **Breadth-First Search**



Q:

15-17: **Breadth-First Search**

- BFS Running time:
  - V vertices
  - E edges

15-18: **Breadth-First Search**

- BFS Running time:
  - V vertices
  - E edges
- Running time $\Theta(V + E)$
  - In terms of just V, $O(V^2)$ (why?)

15-19: **Depth-First Search**

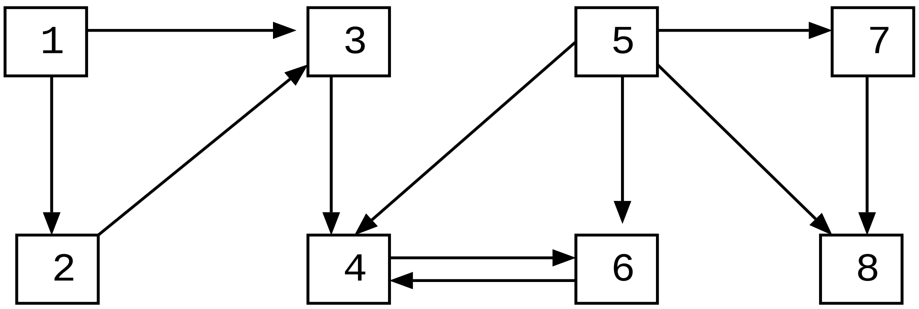
```
DFS(G)
    for each vertex v in G do
        color[v] ← WHITE
        π[v] = nil
    time ← 0
    for each vertex v in G do
        if color[v] = WHITE
            DFS-VISIT(v)
```

15-20: **Depth-First Search**

```
DFS-VISIT(v, G)
    color[v] ← GRAY
    time ← time +1
    d[v] ← time
    for each u adjacent to v in G do
        if color[u] = WHITE then
            π[u] ← v
```

DFS-VISIT(u,G)
color[v] $\leftarrow$ BLACK
time $\leftarrow$ time + 1
$f[v]$ $\leftarrow$ time

15-21: **Depth-First Search**



(Do DFS, show discover/finish times & Depth First Forest)

15-22: **Depth-First Search**

- DFS creates a Depth First Forest
- We can use DFS to classify edges:
  - Tree edges
    - edges in the Depth First Forest
  - Back Edges
    - edge (u, v) that connects u to ancestor v in DFF
  - Forward edges
    - non-tree edge (u, v) that connects u to descendent v in DFF
  - Cross Edges
    - Everything Else

15-23: **Depth-First Search**

- Labeling edges
  - How could we label edges (tree/back/forward/cross) while we are doing DFS?

15-24: **Depth-First Search**

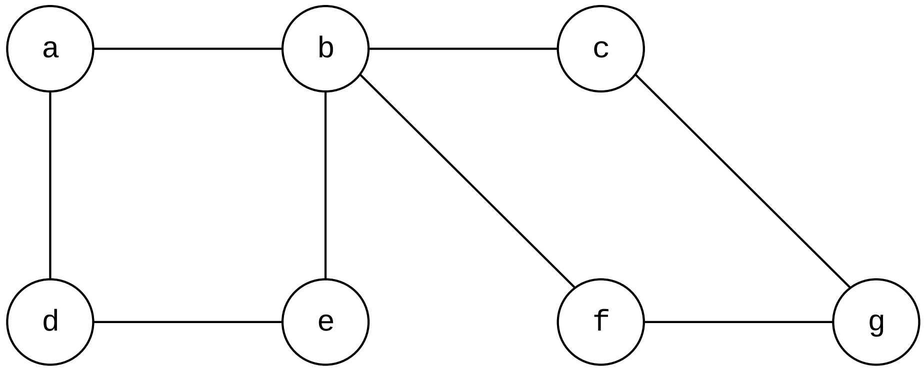
- Labeling edges
  - How could we label edges (tree/back/forward/cross) while we are doing DFS?
  - When examining edge (u, v), if v is:
    - WHITE – tree edge
    - GRAY – back edge
    - BLACK – forward edge or cross edge

15-25: **Depth-First Search**

- Labeling edges

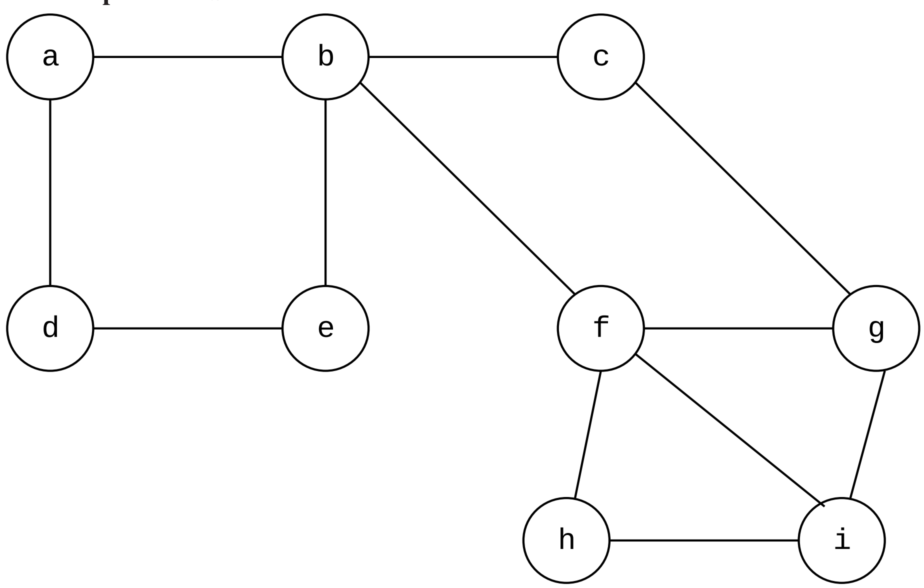
- Can we have cross edges in a DFS of an undirected graph?
- Can we have forward edges in a DFS of an undirected graph?

15-26: **Depth-First Search**



(Do DFS, show discover/finish times & Depth First Forest)

15-27: **Depth-First Search**



(Do DFS, show discover/finish times & Depth First Forest)

15-28: **Topological Sort**

- Directed Acyclic Graph, Vertices $v_1 \dots v_n$
- Create an ordering of the vertices
  - If there a path from v_i to v_j, then v_i appears before v_j in the ordering
- Example: Prerequisite chains

15-29: **Topological Sort**

- How could we use DFS to do a Topological Sort?
  - (Hint – Use discover and/or finish times)

15-30: **Topological Sort**

- How could we use DFS to do a Topological Sort?
  - (Hint – Use discover and/or finish times)
  - (What does it mean if node x finished before node y?)

15-31: **Topological Sort**

- How could we use DFS to do a Topological Sort?
  - Do DFS, computing finishing times for each vertex
  - As each vertex is finished, add to front of a linked list
  - This list is a valid topological sort

15-32: **Topological Sort**

- Second method for doing topological sort:
- Which node(s) could be first in the topological ordering?
  - Node(s) with no incident (incoming) edges

15-33: **Topological Sort**

- Pick a node v_k with no incident edges
- Add v_k to the ordering
- Remove v_k and all edges from v_k from the graph
- Repeat until all nodes are picked.

15-34: **Topological Sort**

- How can we find a node with no incident edges?
- Count the incident edges of all nodes

15-35: **Topological Sort**

```
for (i=0; i < NumberOfVertices; i++)
   NumIncident[i] = 0;

for(i=0; i < NumberOfVertices; i++)
   each node k adjacent to i
      NumIncident[k]++
```

15-36: **Topological Sort**

```
for(i=0; i < NumberOfVertices; i++)
   NumIncident[i] = 0;

for(i=0; i < NumberOfVertices; i++)
  for(tmp=G[i]; tmp != null; tmp=tmp.next())
    NumIncident[tmp.neighbor()]++
```

15-37: **Topological Sort**

- Create NumIncident array
- Repeat
  - Search through NumIncident to find a vertex v with NumIncident[v] == 0
  - Add v to the ordering
  - Decrement NumIncident of all neighbors of v
  - Set NumIncident[v] = -1
- Until all vertices have been picked

15-38: **Topological Sort**

- In a graph with V vertices and E edges, how long does this version of topological sort take?

15-39: **Topological Sort**

- In a graph with V vertices and E edges, how long does this version of topological sort take?
  - $\Theta(V^2 + E) = \Theta(V^2)$
    - Since $E \in O(V^2)$

15-40: **Topological Sort**

- Where are we spending “extra” time

15-41: **Topological Sort**

- Where are we spending “extra” time
  - Searching through NumIncident each time looking for a vertex with no incident edges
  - Keep around a set of all nodes with no incident edges
  - Remove an element v from this set, and add it to the ordering
  - Decrement NumIncident for all neighbors of v
    - If NumIncident[k] is decremented to 0, add k to the set.
  - How do we implement the set of nodes with no incident edges?

15-42: **Topological Sort**

- Where are we spending “extra” time
  - Searching through NumIncident each time looking for a vertex with no incident edges
  - Keep around a set of all nodes with no incident edges
  - Remove an element v from this set, and add it to the ordering
  - Decrement NumIncident for all neighbors of v
    - If NumIncident[k] is decremented to 0, add k to the set.
  - How do we implement the set of nodes with no incident edges?
    - Use a stack

15-43: **Topological Sort**

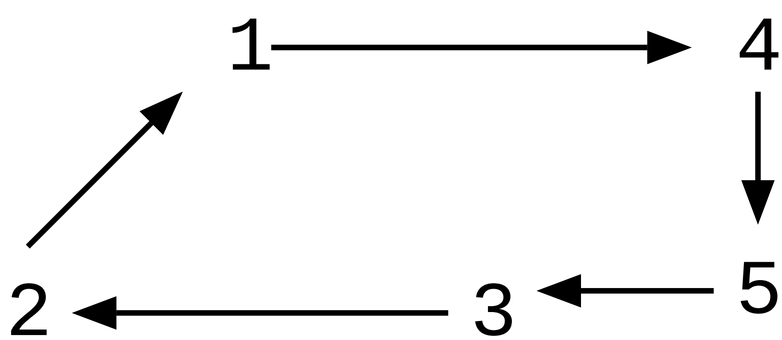
- Examples!!
  - Graph
  - Adjacency List
  - NumIncident
  - Stack

15-44: **More DFS Applications**

- Depth First Search can be used to calculate the connected components of a directed graph
- First, some definitions and examples:

15-45: **Strongly Connected Graph**

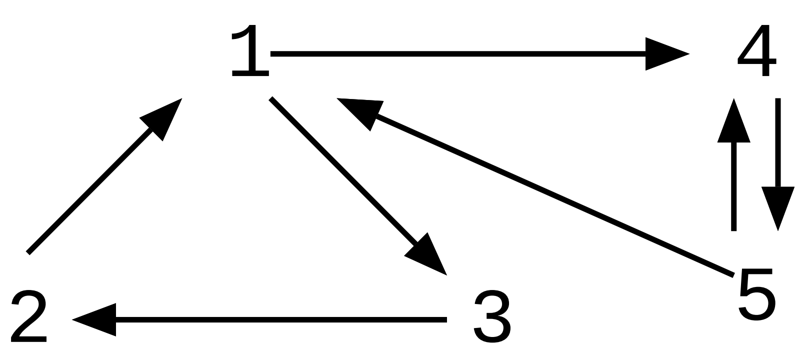
- Directed Path from every node to every other node



- Strongly Connected

15-46: **Strongly Connected Graph**

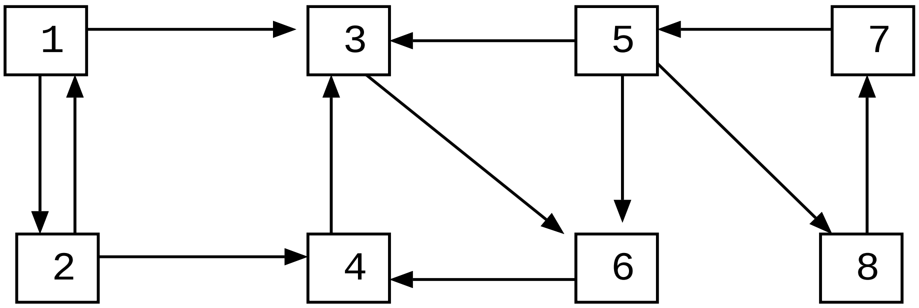
- Directed Path from every node to every other node



- Strongly Connected

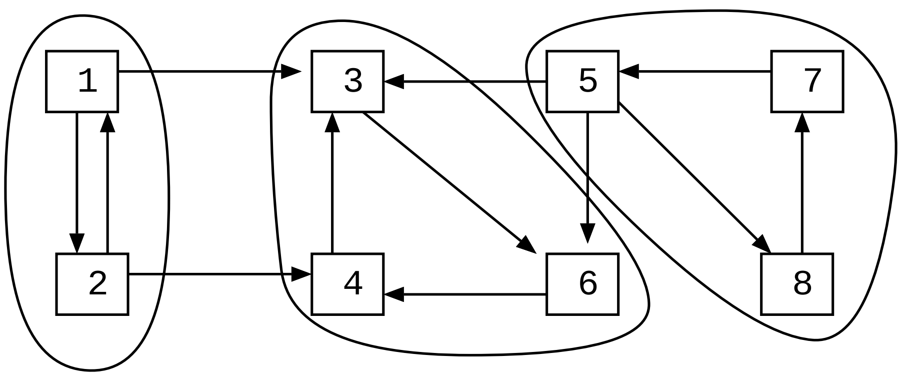
15-47: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.



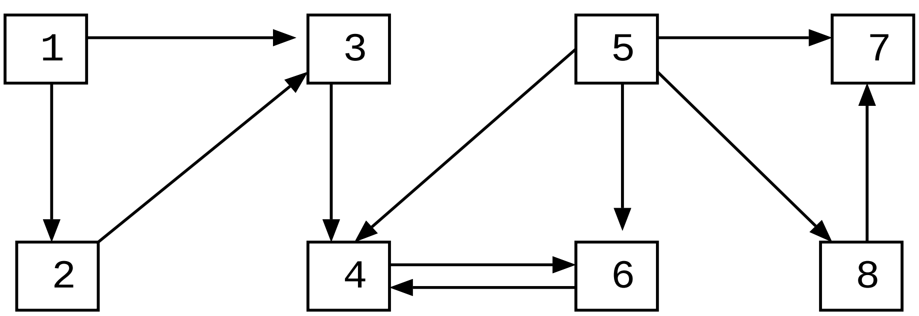
15-48: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.



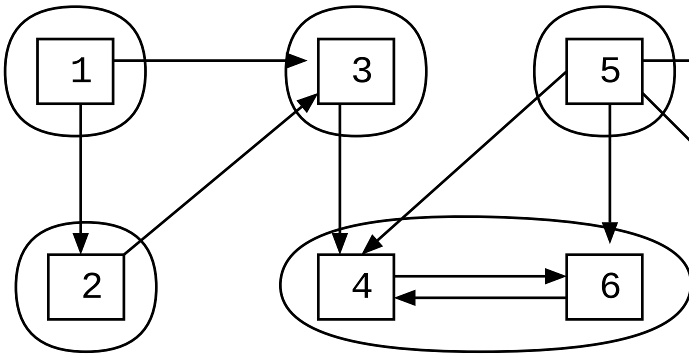
15-49: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.



15-50: **Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.



15-51: **Connected Components**

- Connected components of the graph are the *largest possible* strongly connected subgraphs
- If we put each vertex in its own component – each component would be (trivially) strongly connected
  - Those would not be the connected components of the graph – unless there were no larger connected subgraphs

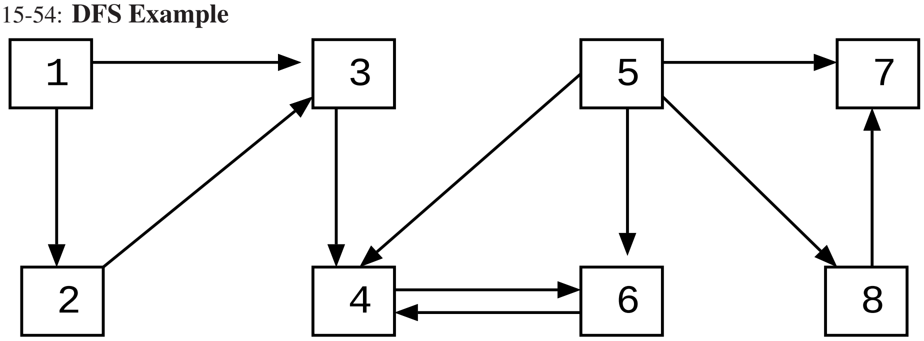
15-52: **Connected Components**

- Calculating Connected Components
  - Two vertices v_1 and v_2 are in the same connected component if and only if:
    - Directed path from v_1 to v_2
    - Directed path from v_2 to v_1
  - To find connected components – find directed paths
    - Use DFS: $d[v]$ and $f[v]$

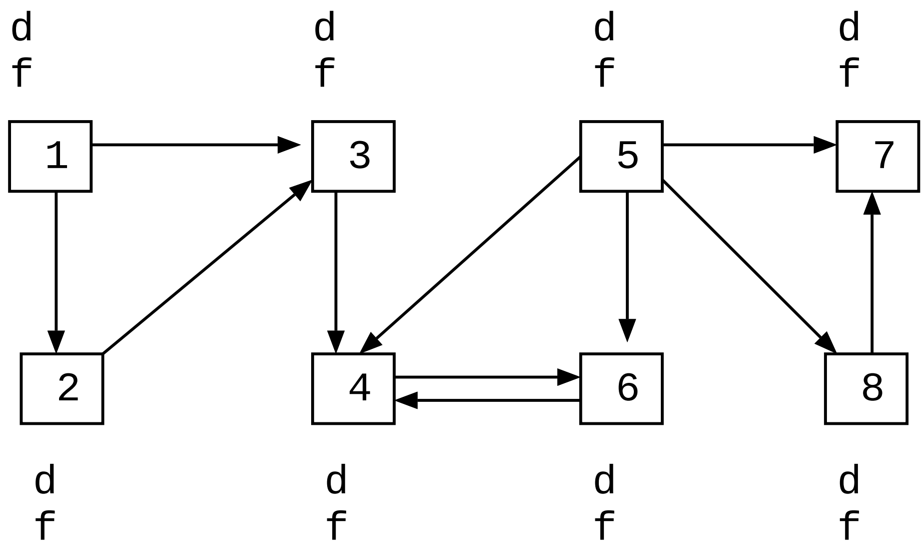
15-53: **DFS Revisited**

- Recall that we calculate the order in which we visit the elements in a Depth-First Search
- For any vertex v in a DFS:
  - d[v] = *Discovery* time – when the vertex is first visited
  - f[v] = *Finishing* time – when we have finished with a vertex (and all of its children)

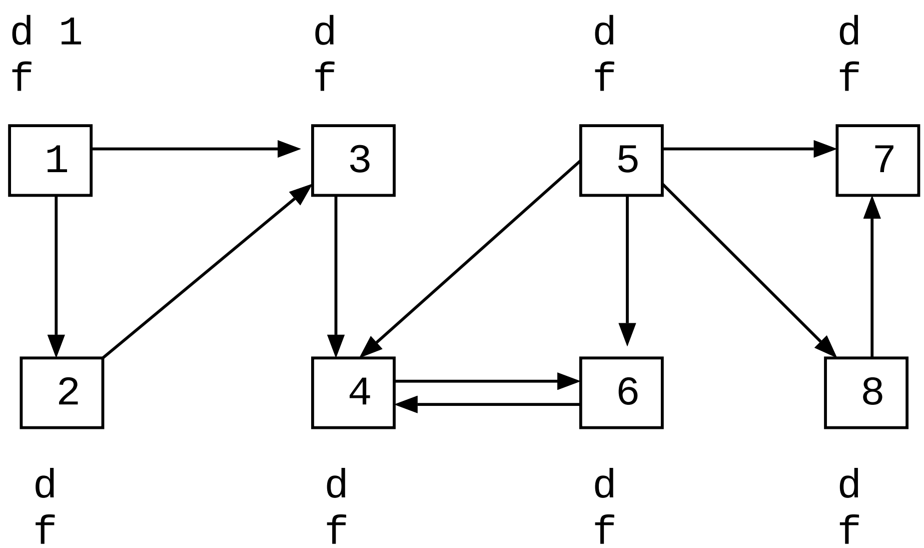
15-54: **DFS Example**



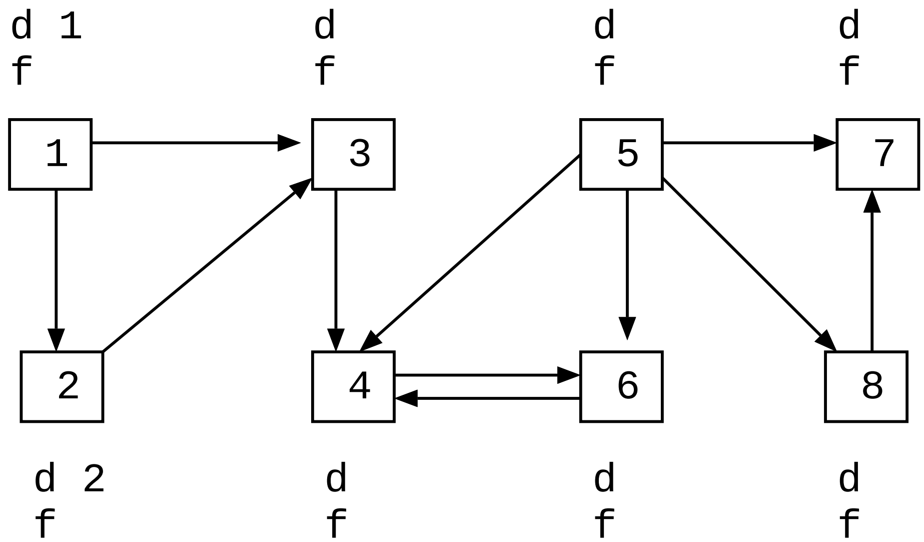
15-55: **DFS Example**



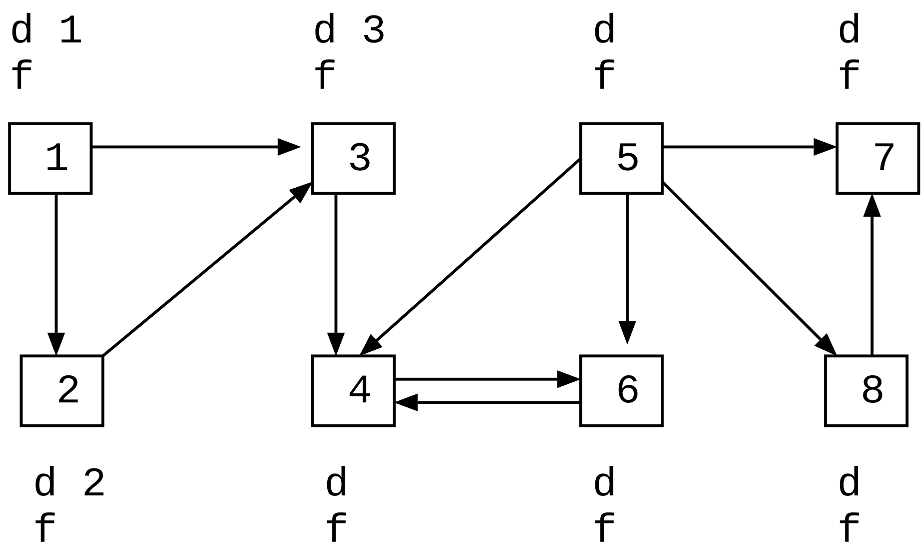
15-56: **DFS Example**



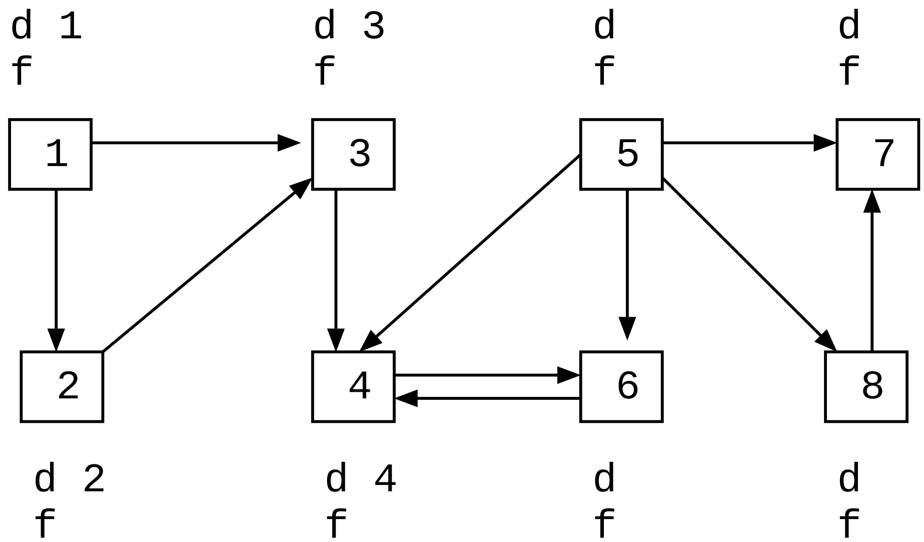
15-57: **DFS Example**



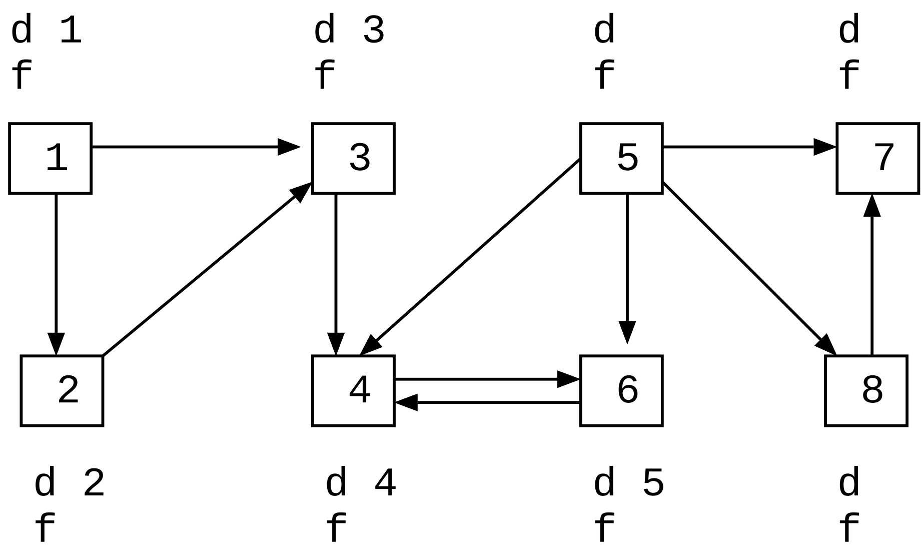
15-58: **DFS Example**



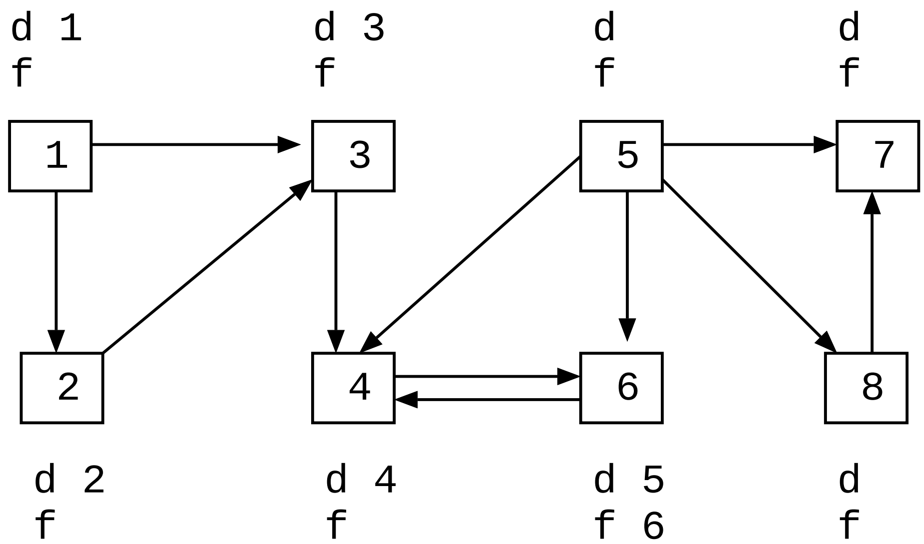
15-59: **DFS Example**



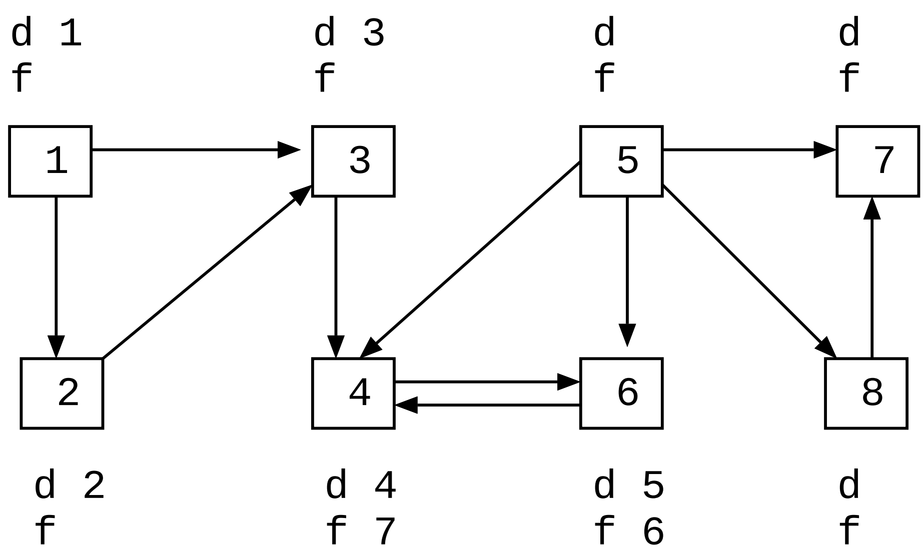
15-60: **DFS Example**



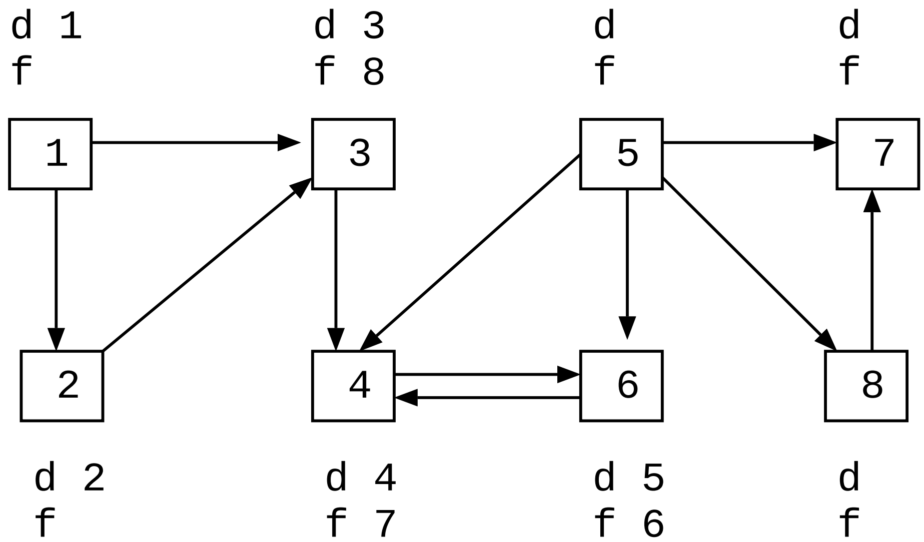
15-61: **DFS Example**



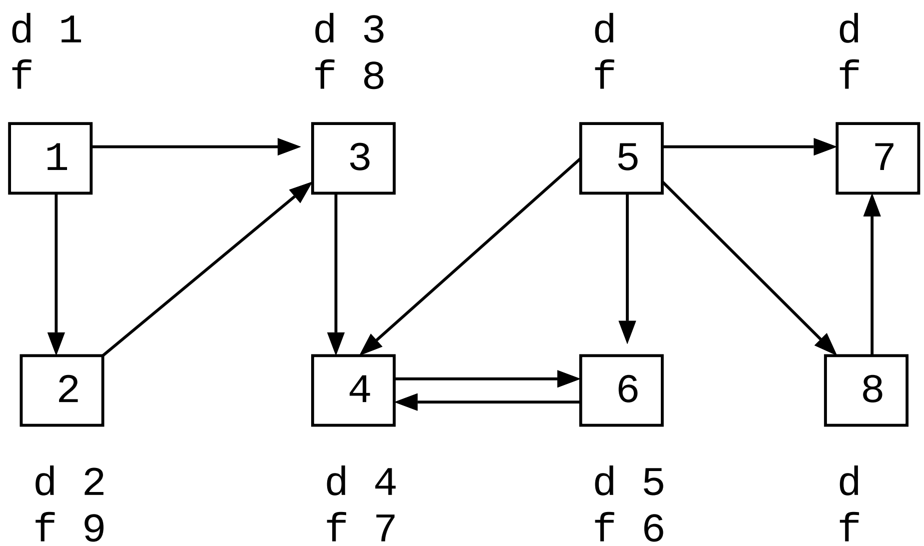
15-62: **DFS Example**



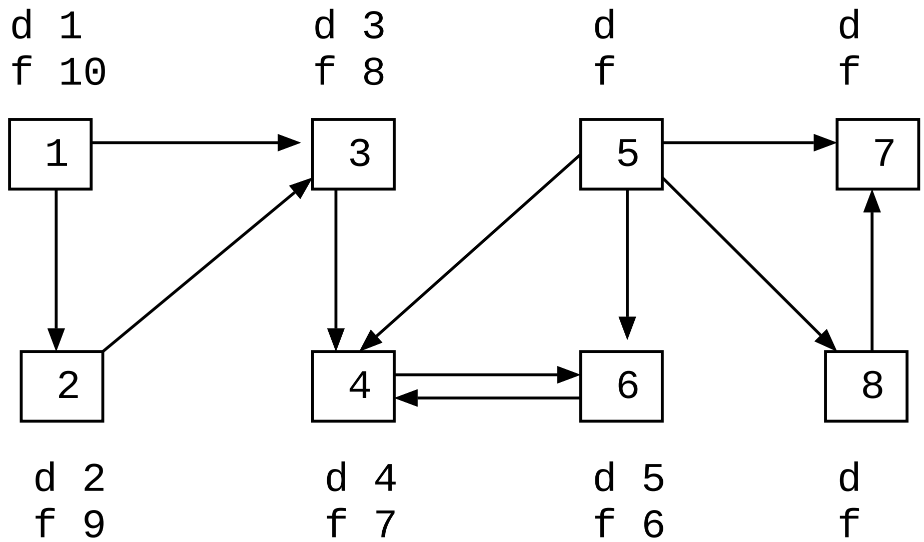
15-63: **DFS Example**



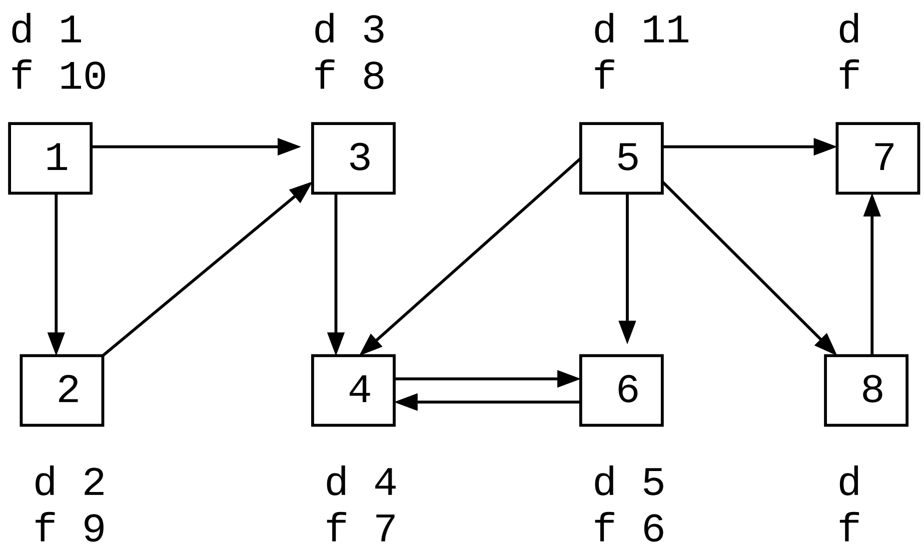
15-64: **DFS Example**



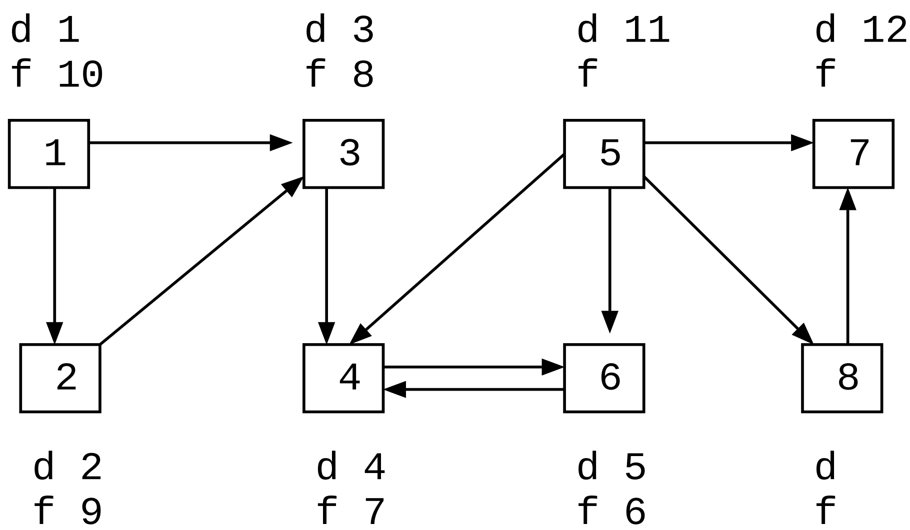
15-65: **DFS Example**



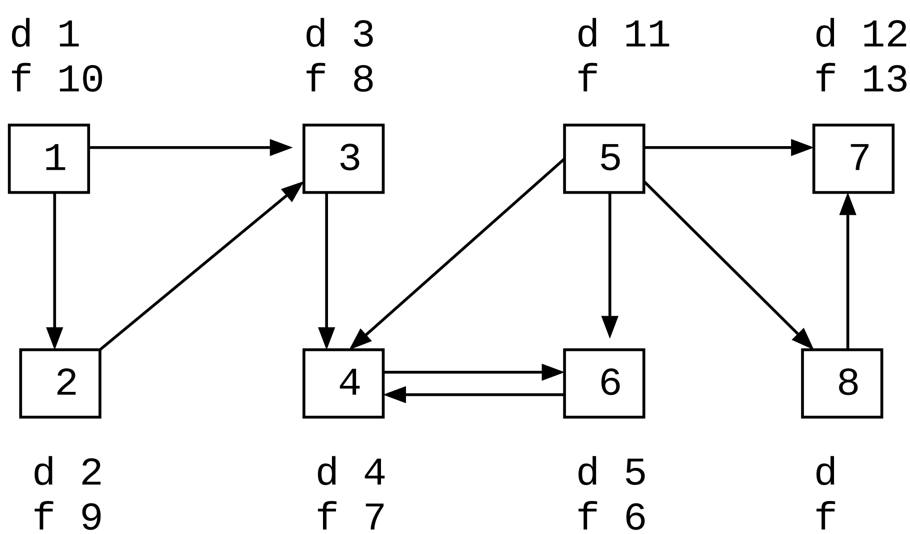
15-66: **DFS Example**



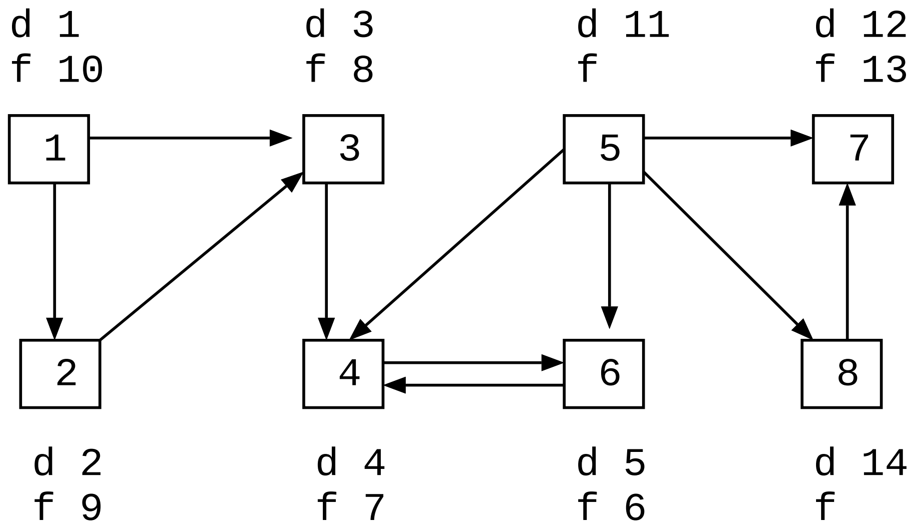
15-67: **DFS Example**



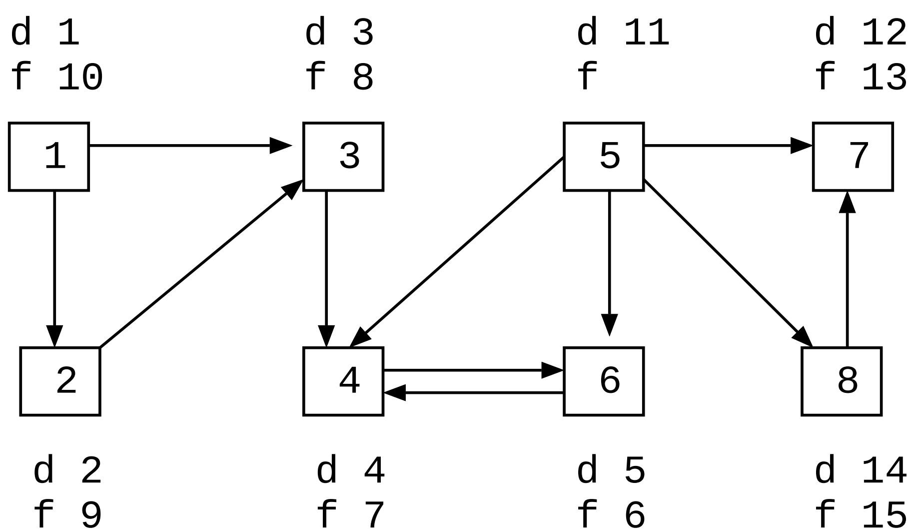
15-68: **DFS Example**



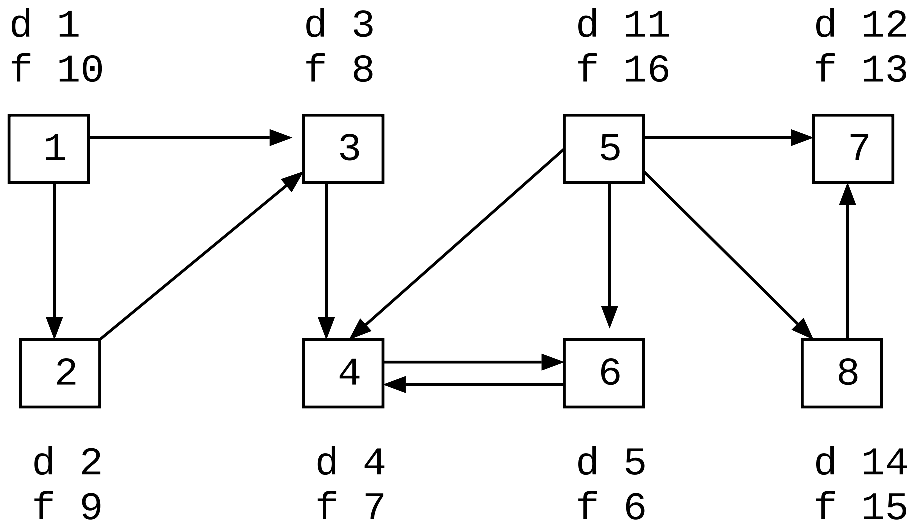
15-69: **DFS Example**



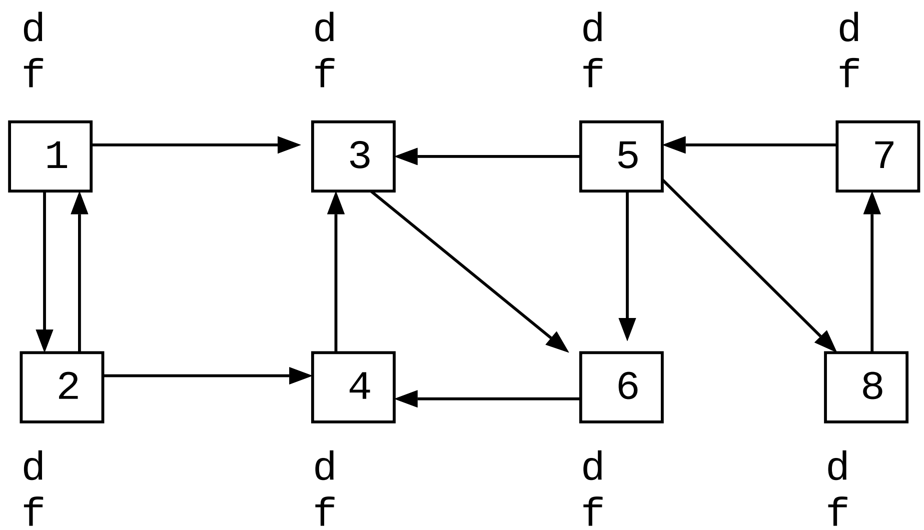
15-70: **DFS Example**



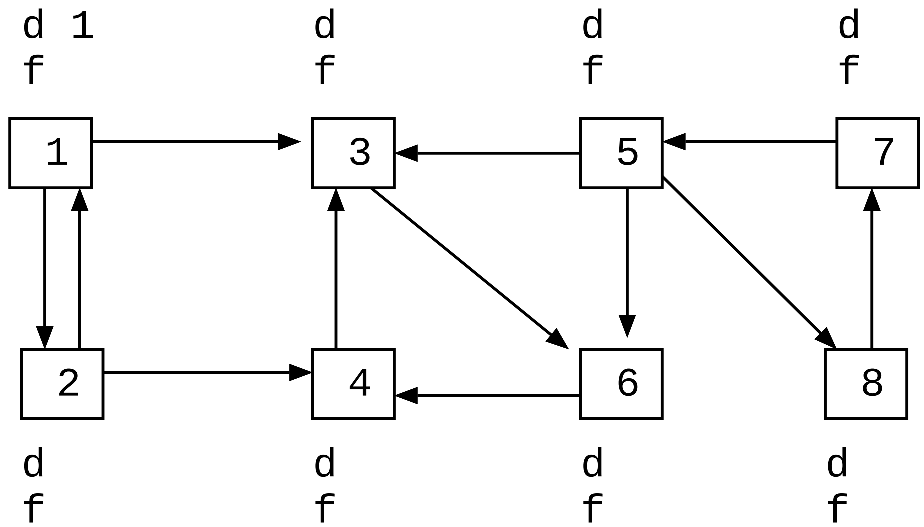
15-71: **DFS Example**



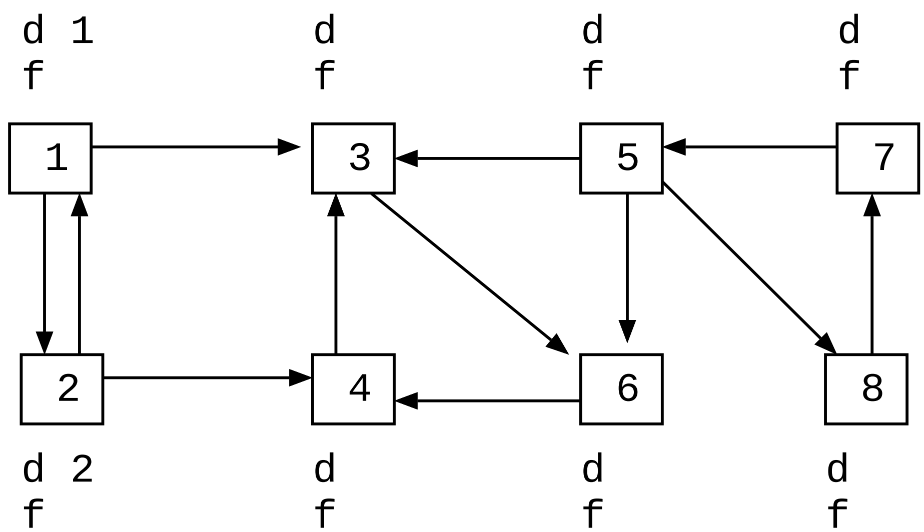
15-72: **DFS Example**



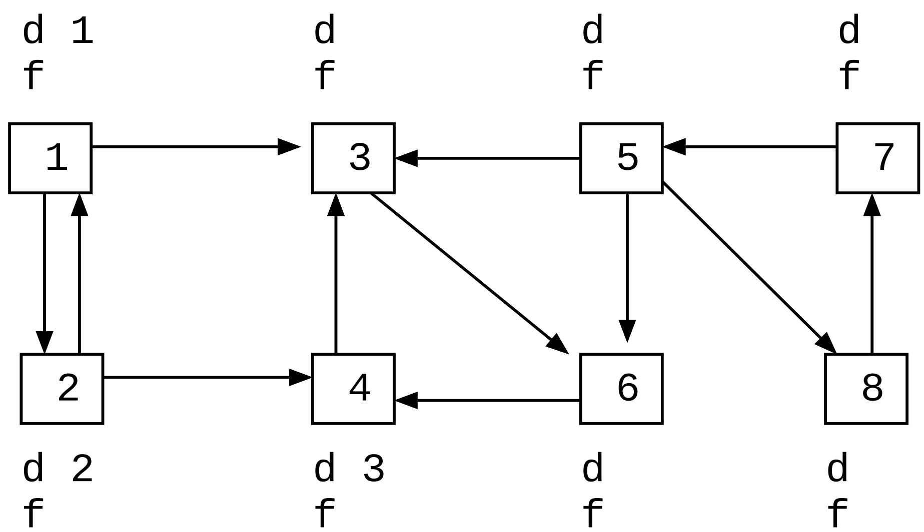
15-73: **DFS Example**



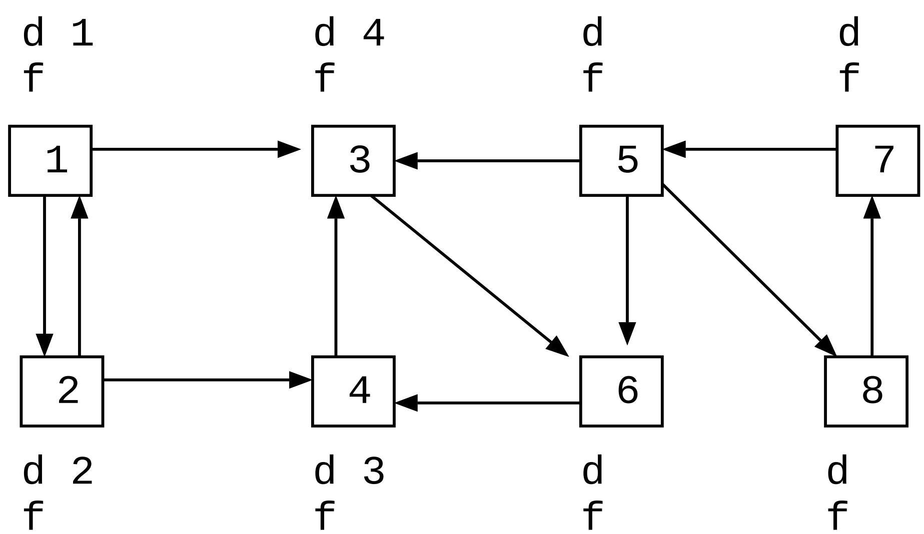
15-74: **DFS Example**



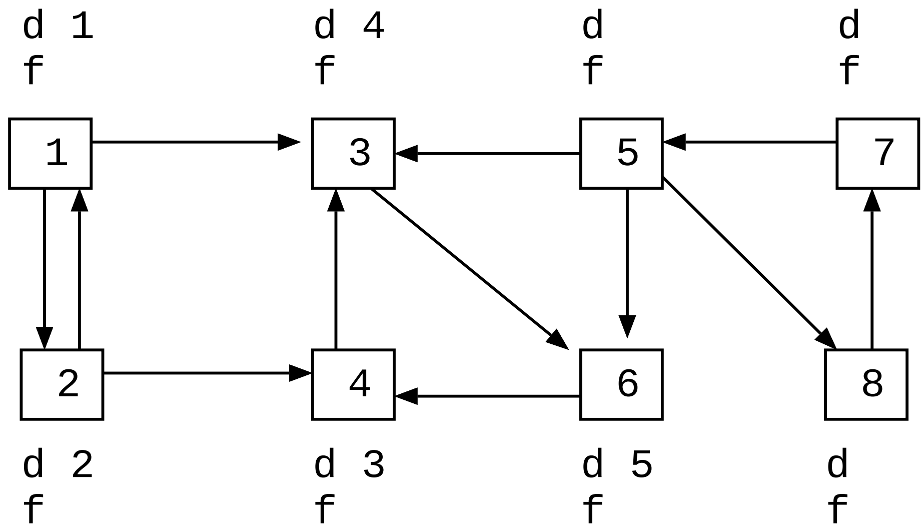
15-75: **DFS Example**



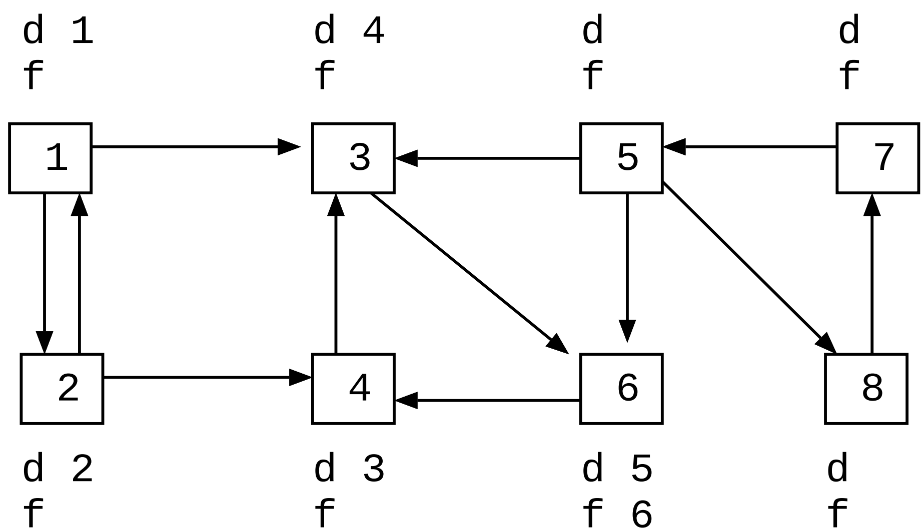
15-76: **DFS Example**



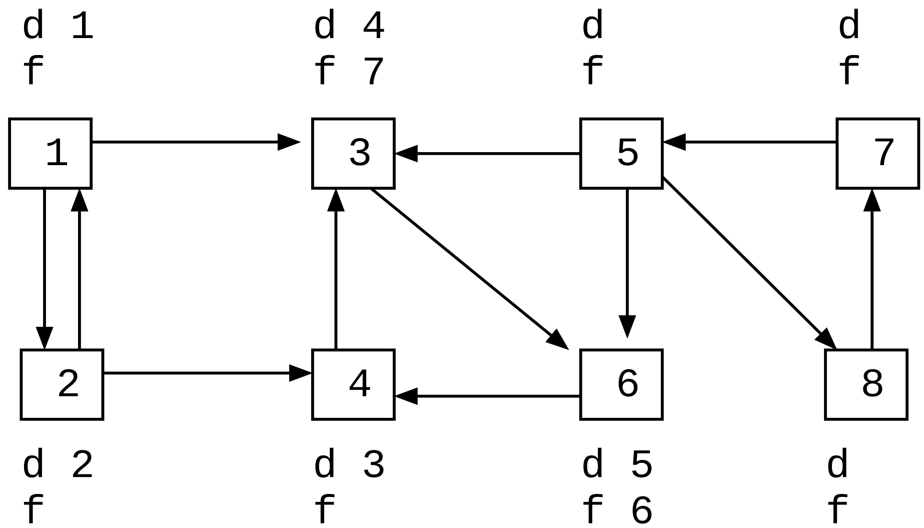
15-77: **DFS Example**



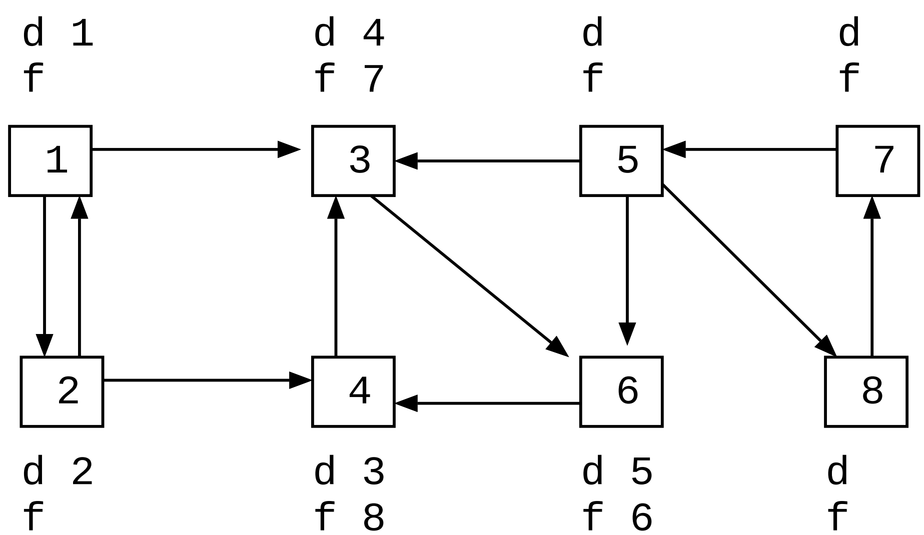
15-78: **DFS Example**



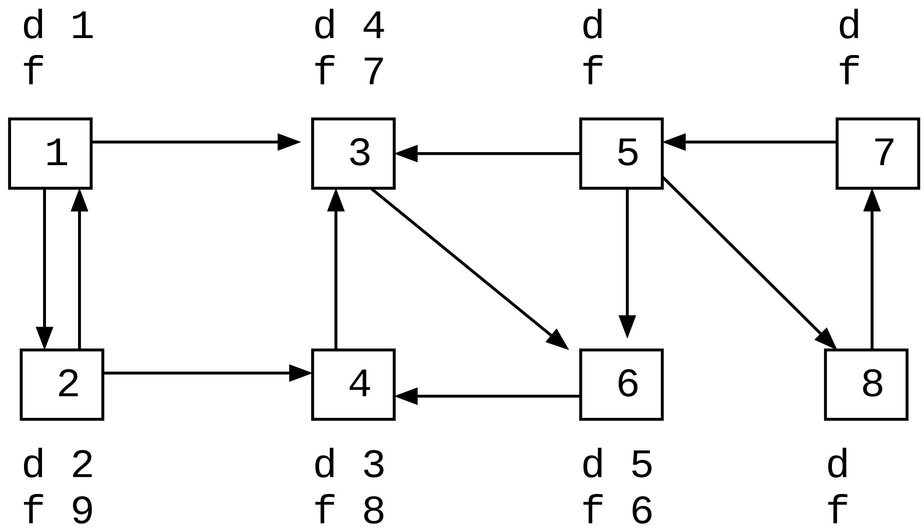
15-79: **DFS Example**



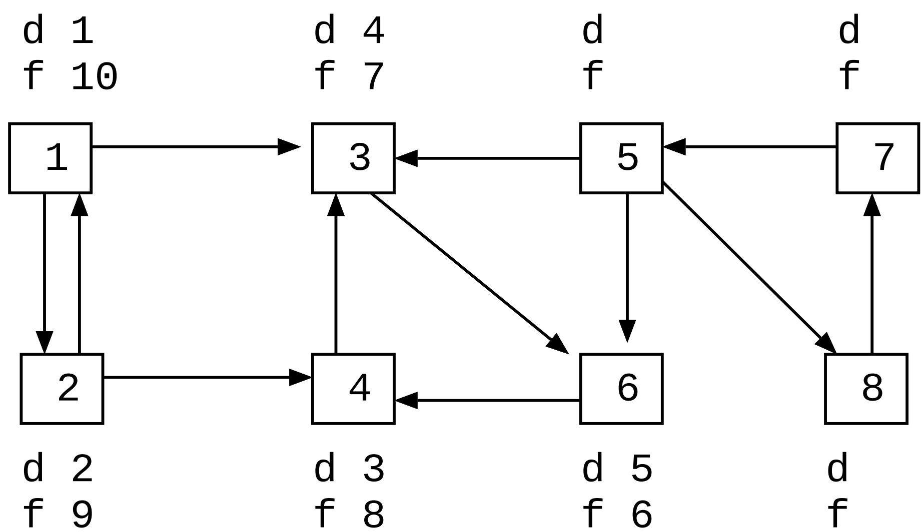
15-80: **DFS Example**



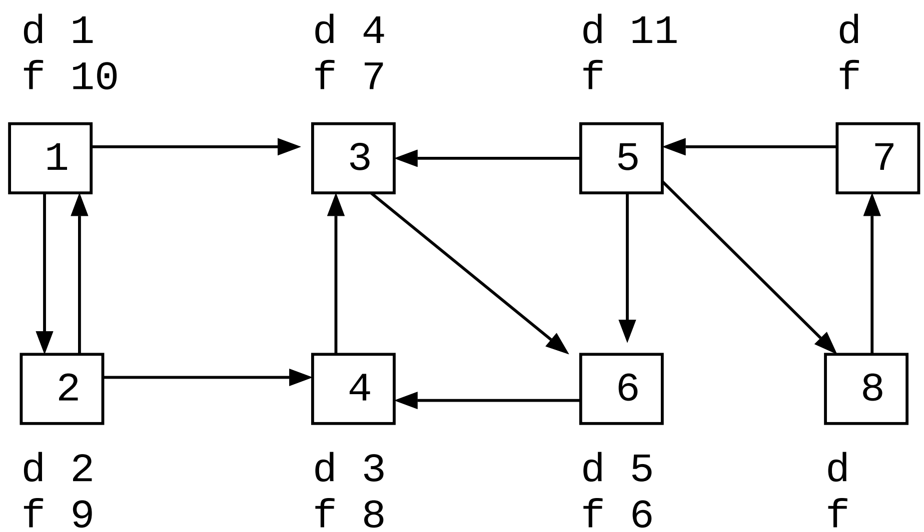
15-81: **DFS Example**



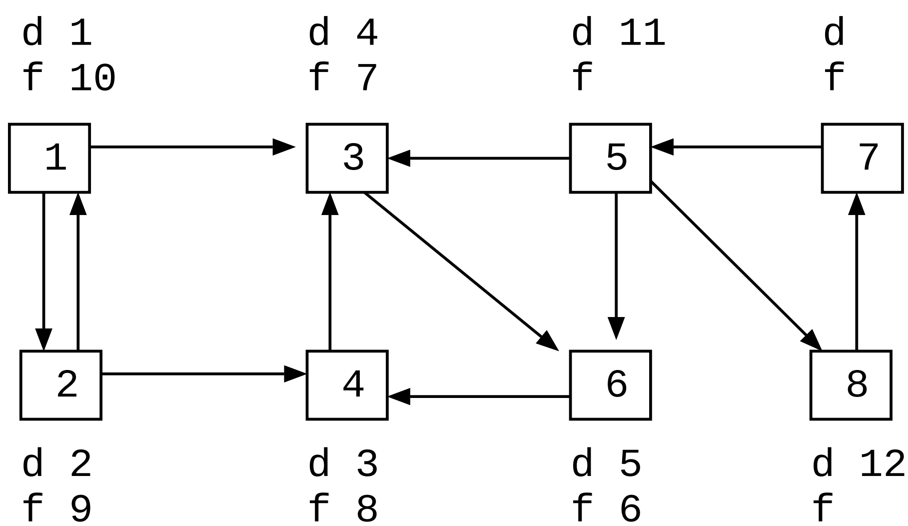
15-82: **DFS Example**



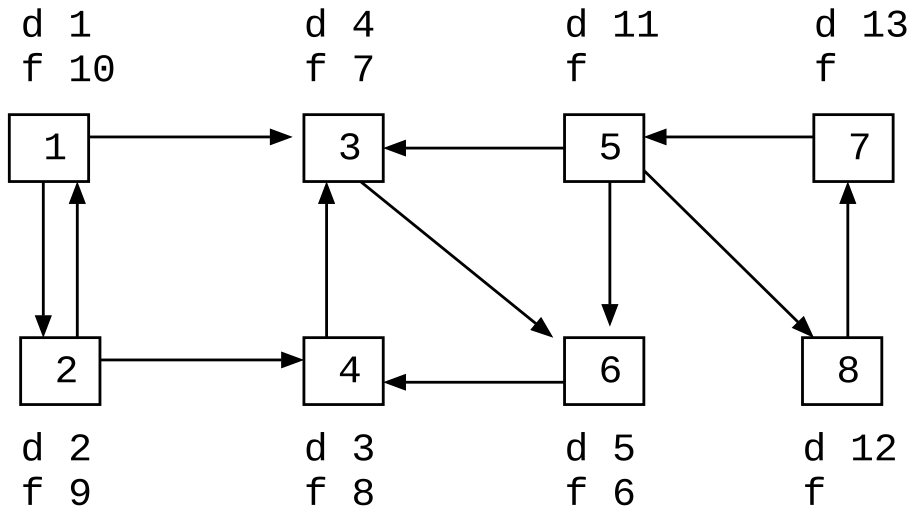
15-83: **DFS Example**



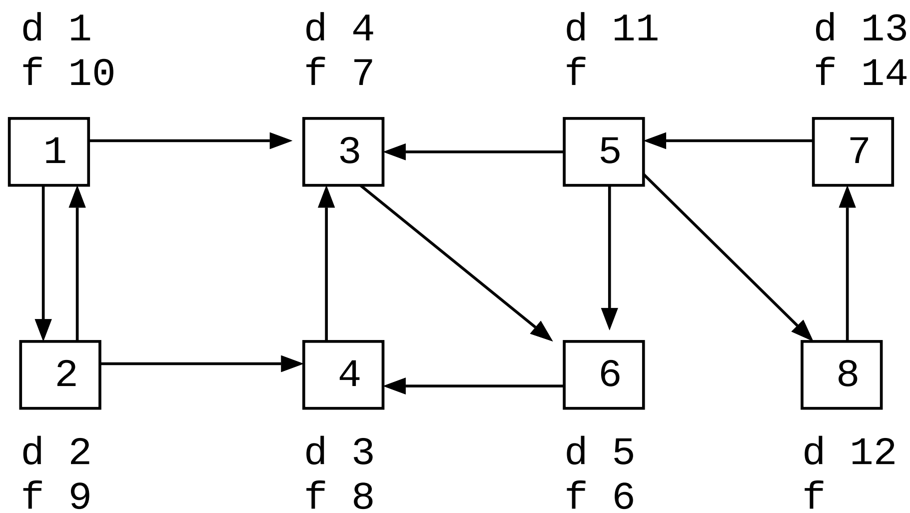
15-84: **DFS Example**



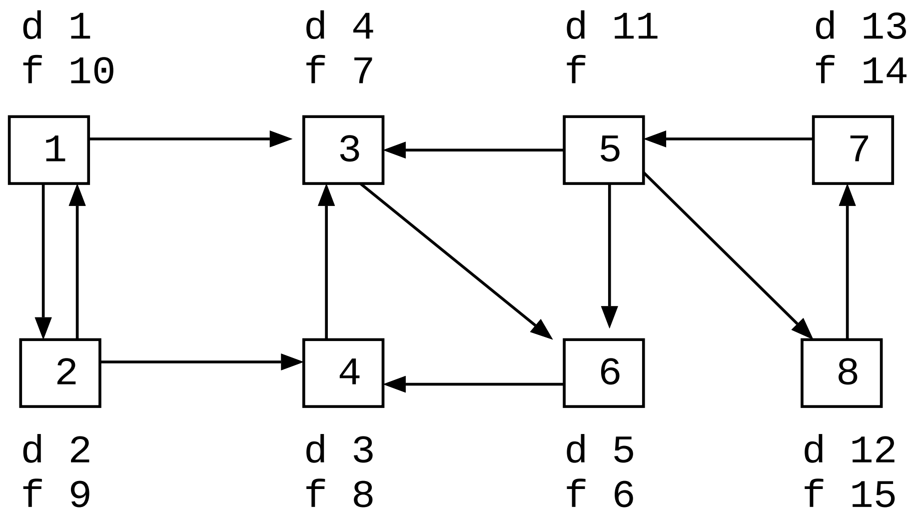
15-85: **DFS Example**



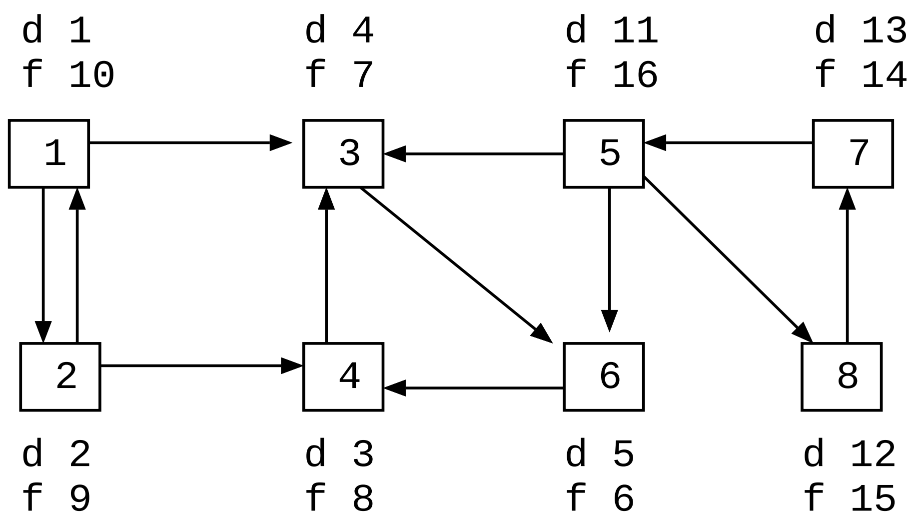
15-86: **DFS Example**



15-87: **DFS Example**



15-88: **DFS Example**



15-89: **Using d[] & f[]**

- Given two vertices v_1 and v_2, what do we know if $f[v_2] < f[v_1]$?

15-90: **Using d[] & f[]**

- Given two vertices v_1 and v_2, what do we know if $f[v_2] < f[v_1]$?
  - Either:
    - Path from v_1 to v_2
      - Start from v_1
      - Eventually visit v_2
      - Finish v_2
      - Finish v_1

15-91: **Using d[] & f[]**

- Given two vertices v_1 and v_2, what do we know if $f[v_2] < f[v_1]$?
  - Either:
    - Path from v_1 to v_2
    - No path from v_2 to v_1
      - Start from v_2
      - Eventually finish v_2
      - Start from v_1
      - Eventually finish v_1

15-92: **Using d[] & f[]**

- If $f[v_2] < f[v_1]$:
  - Either a path from v_1 to v_2, or no path from v_2 to v_1
  - If there is a path from v_2 to v_1, then there must be a path from v_1 to v_2
- $f[v_2] < f[v_1]$ and a path from v_2 to v_1 $\Rightarrow$ v_1 and v_2 are in the same connected component

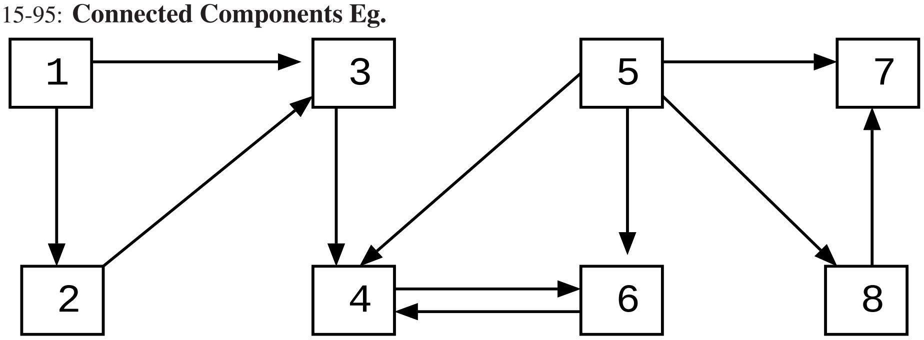
15-93: **Calculating paths**

- Path from v_2 to v_1 in G if and only if there is a path from v_1 to v_2 in G^T
  - G^T is the transpose of G – G with all edges reversed
- If after DFS, $f[v_2] < f[v_1]$
- Run second DFS on G^T, starting from v_1, and v_1 and v_2 are in the same DFS spanning tree
- v_1 and v_2 must be in the same connected component

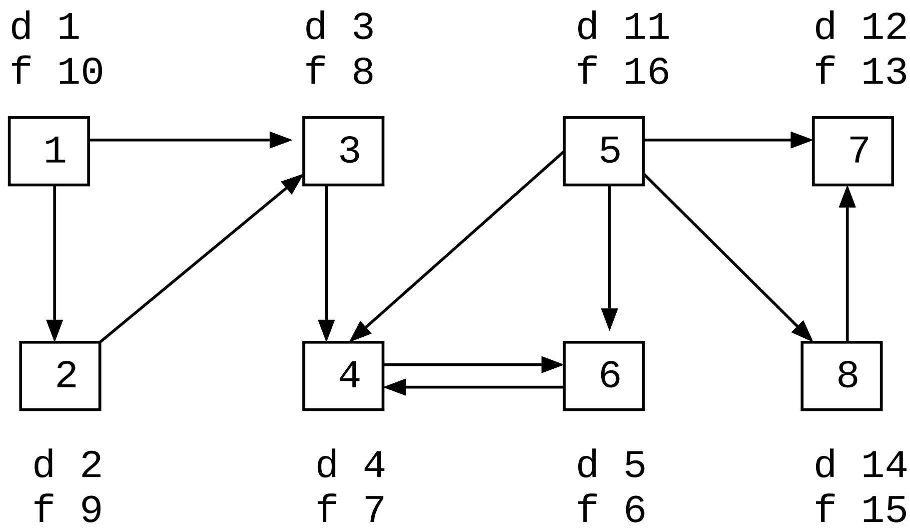
15-94: **Connected Components**

- Run DFS on G, calculating f[] times
- Compute G^T
- Run DFS on G^T – examining nodes in *inverse order of finishing times* from first DFS
- Any nodes that are in the same DFS search tree in G^T must be in the same connected component

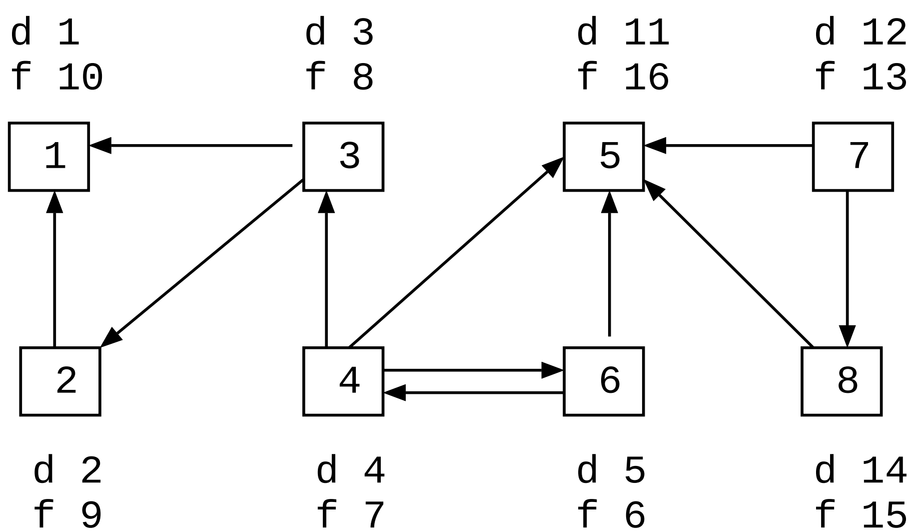
15-95: **Connected Components Eg.**



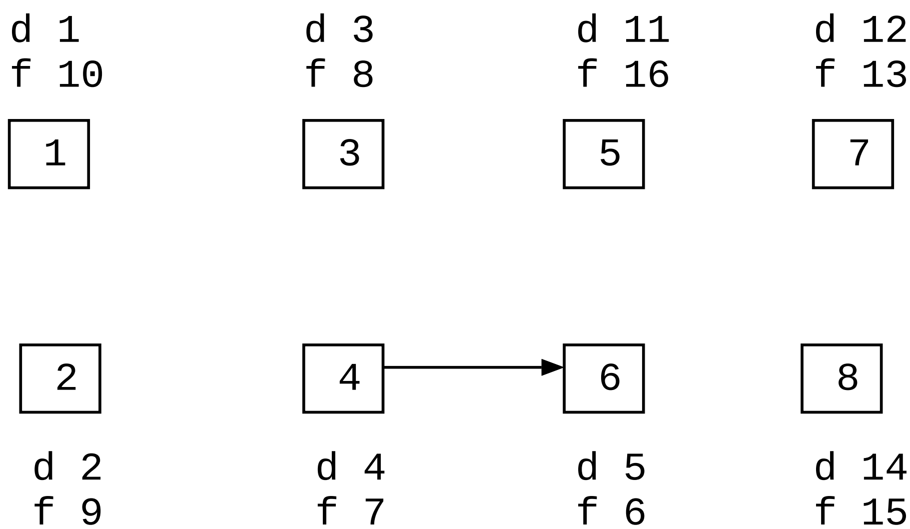
15-96: **Connected Components Eg.**



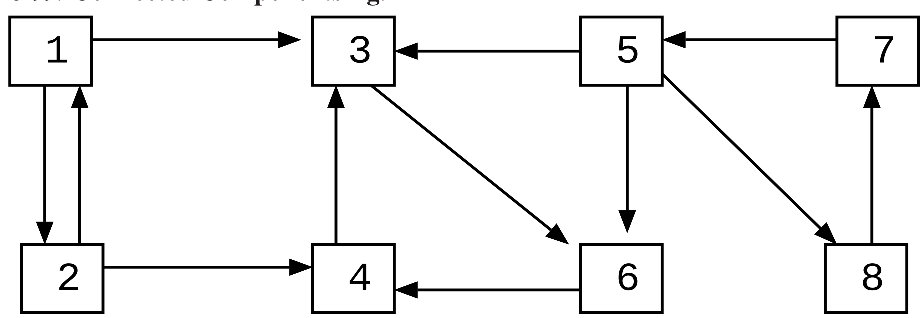
15-97: **Connected Components Eg.**



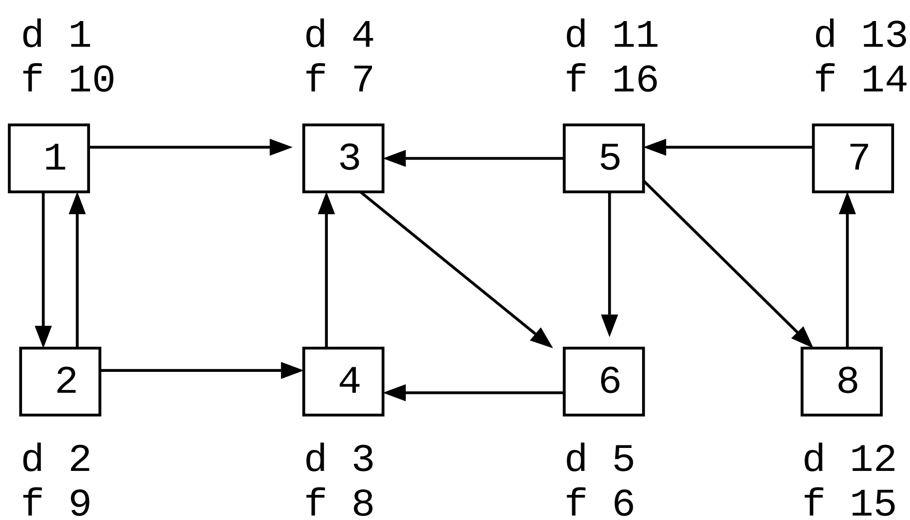
15-98: **Connected Components Eg.**



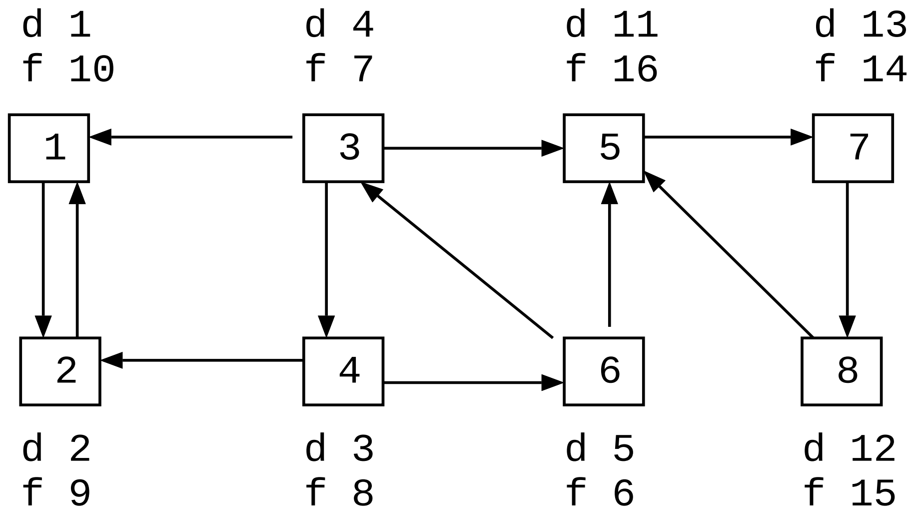
15-99: **Connected Components Eg.**



15-100: **Connected Components Eg.**



15-101: **Connected Components Eg.**



15-102: **Connected Components Eg.**