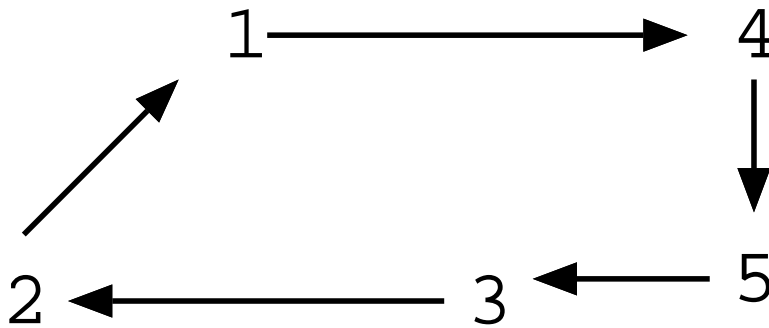


21-0: Strongly Connected Graph

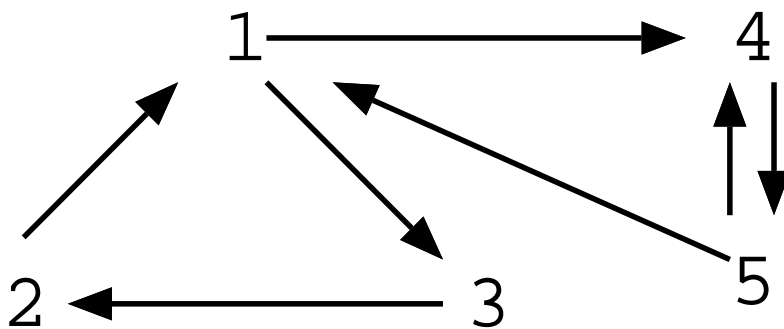
- Directed Path from every node to every other node



- Strongly Connected

21-1: Strongly Connected Graph

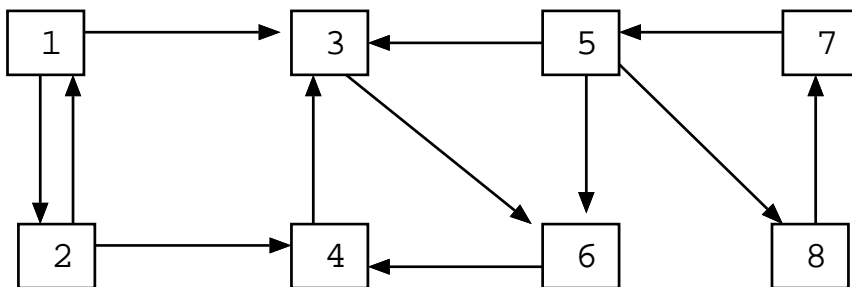
- Directed Path from every node to every other node



- Strongly Connected

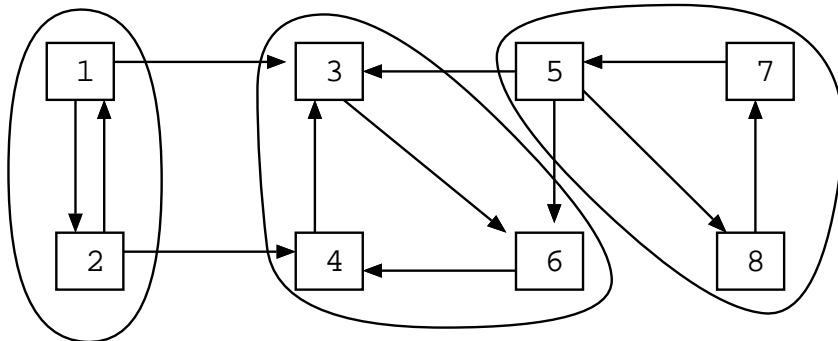
21-2: Connected Components

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



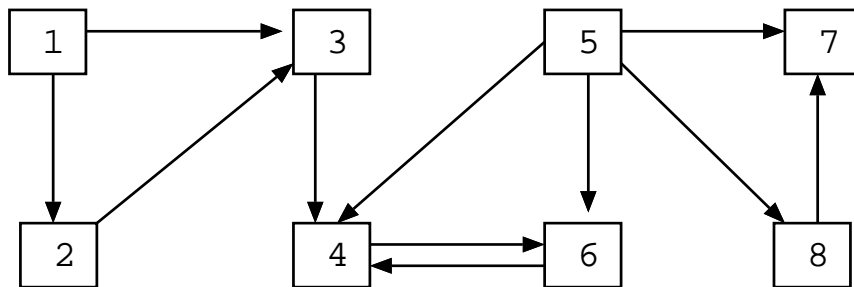
21-3: Connected Components

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



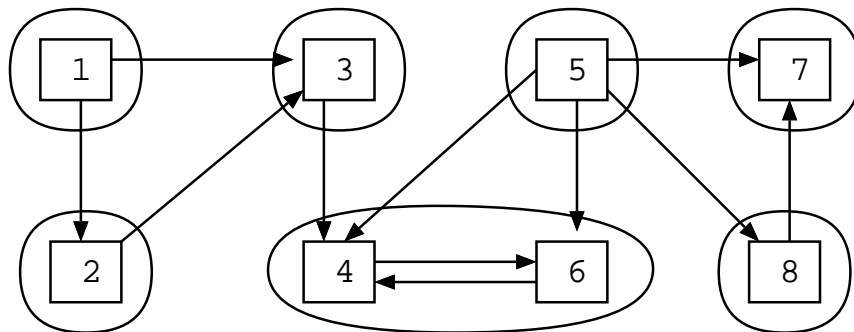
21-4: Connected Components

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



21-5: Connected Components

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



21-6: 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

21-7: 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

21-8: DFS Revisited

- We can keep track of 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)

21-9: DFS Revisited

```
class Edge {
    public int neighbor;
    public int next;
}

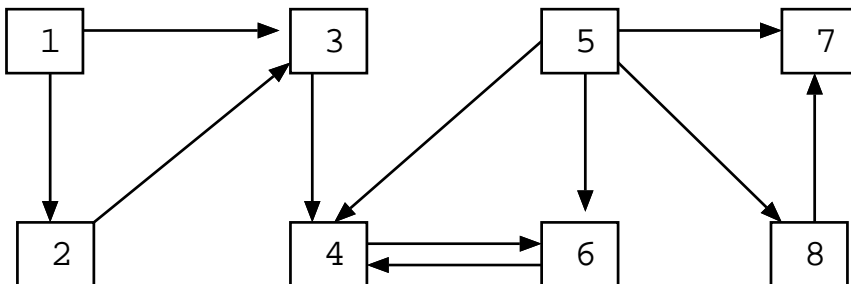
void DFS(Edge G[], int vertex, boolean Visited[], int d[], int f[]) {
    Edge tmp;
    Visited[vertex] = true;
    d[vertex] = time++;
    for (tmp = G[vertex]; tmp != null; tmp = tmp.next) {
        if (!Visited[tmp.neighbor])
            DFS(G, tmp.neighbor, Visited);
    }
    f[vertex] = time++;
}
```

21-10: DFS Revisited

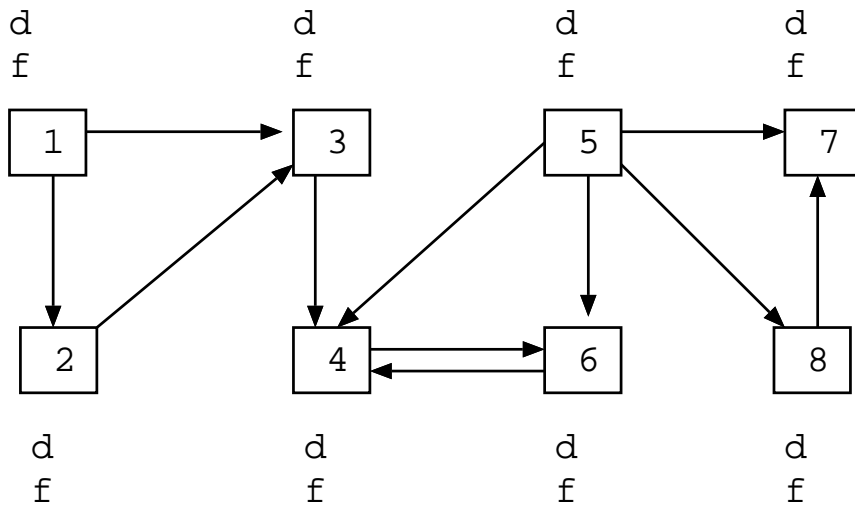
- To visit every node in the graph:

```
TraverseDFS(Edge G[]) {
    int i;
    boolean Visited = new boolean[G.length];
    int d = new int[G.length];
    int v = new int[G.length];
    time = 1;
    for (i=0; i<G.length; i++)
        Visited[i] = false;
    for (i=0; i<G.length; i++)
        if (!Visited[i])
            DFS(G, i, Visited, d, f);
}
```

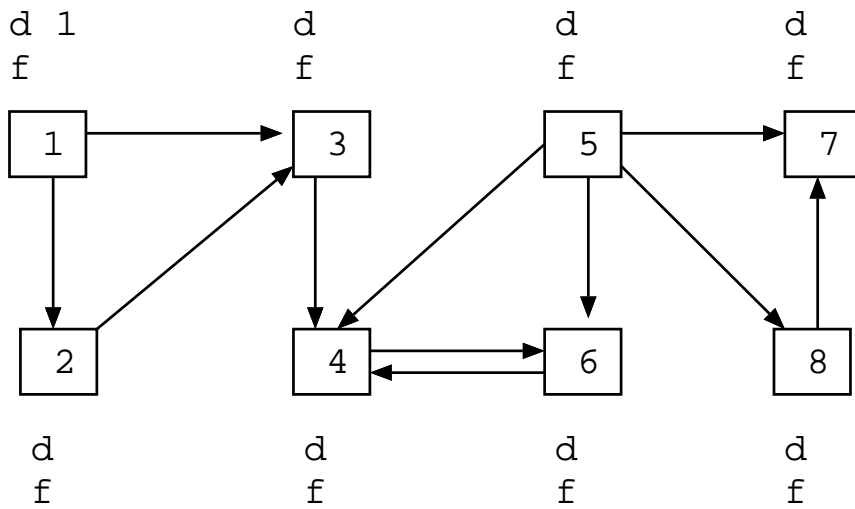
21-11: DFS Example



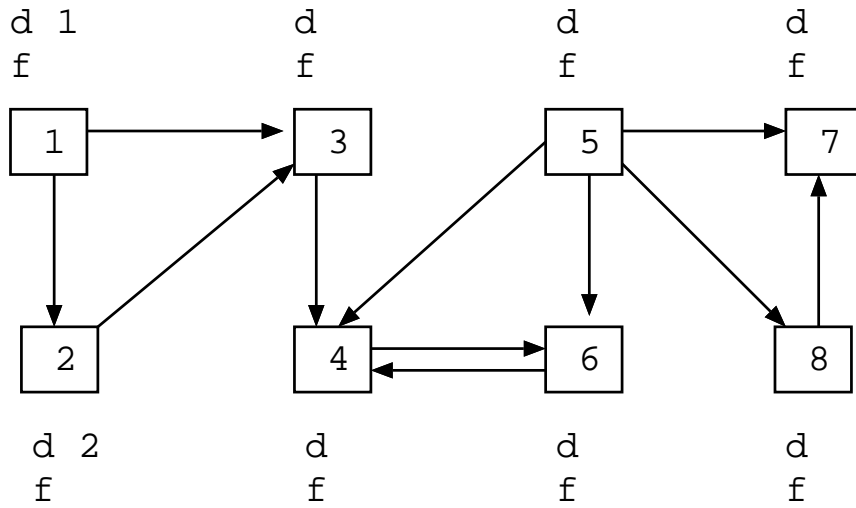
21-12: DFS Example



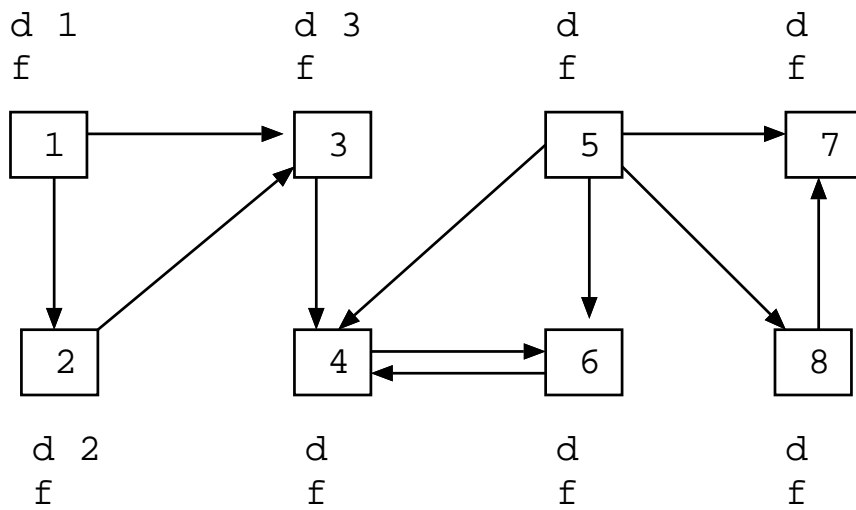
21-13: DFS Example



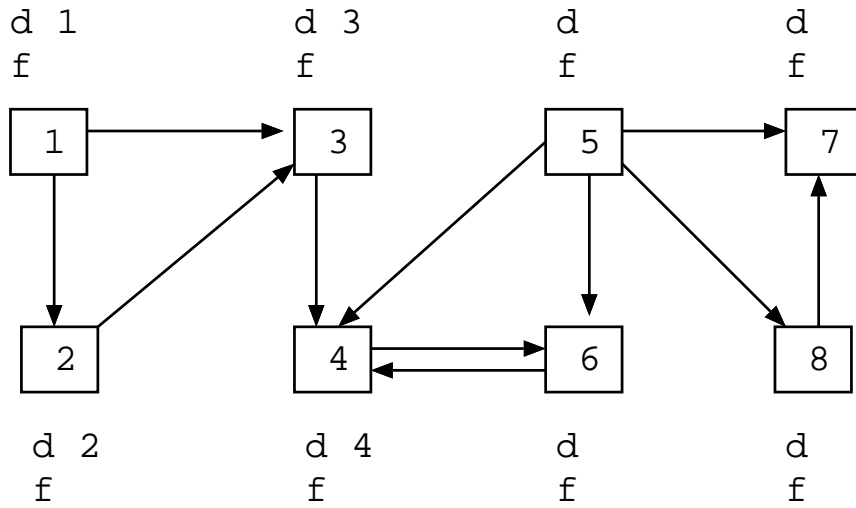
21-14: DFS Example



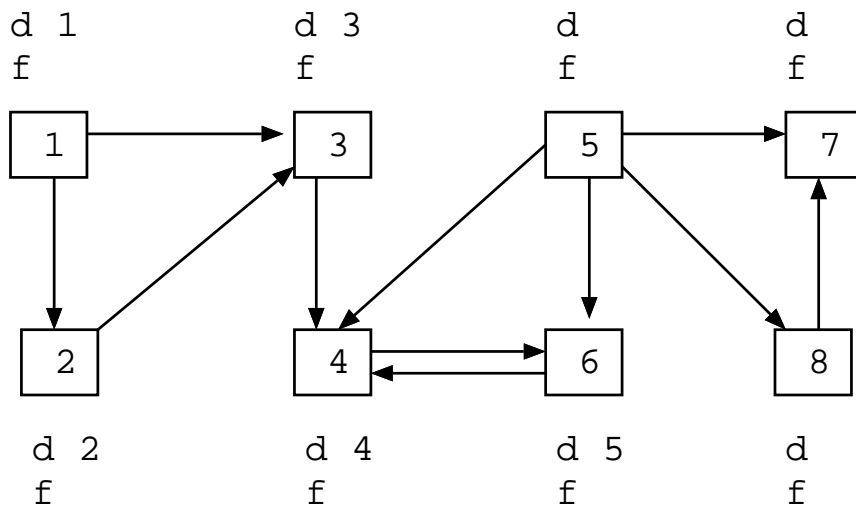
21-15: DFS Example



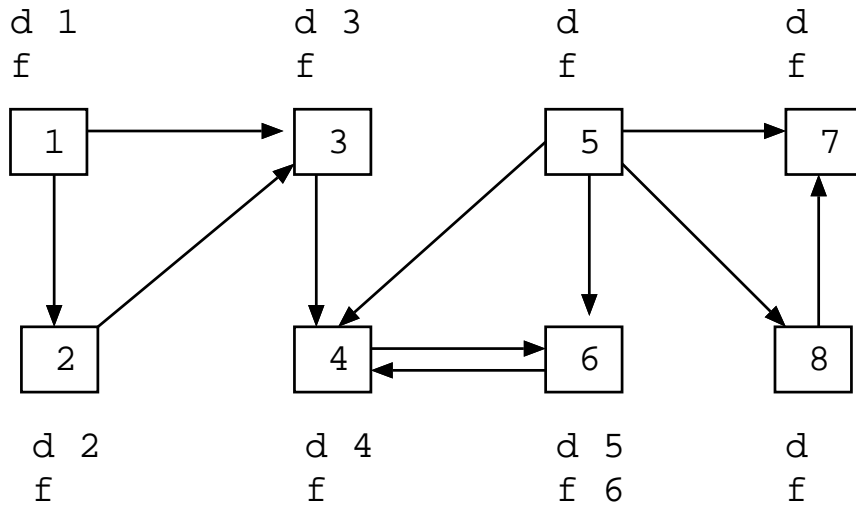
21-16: DFS Example



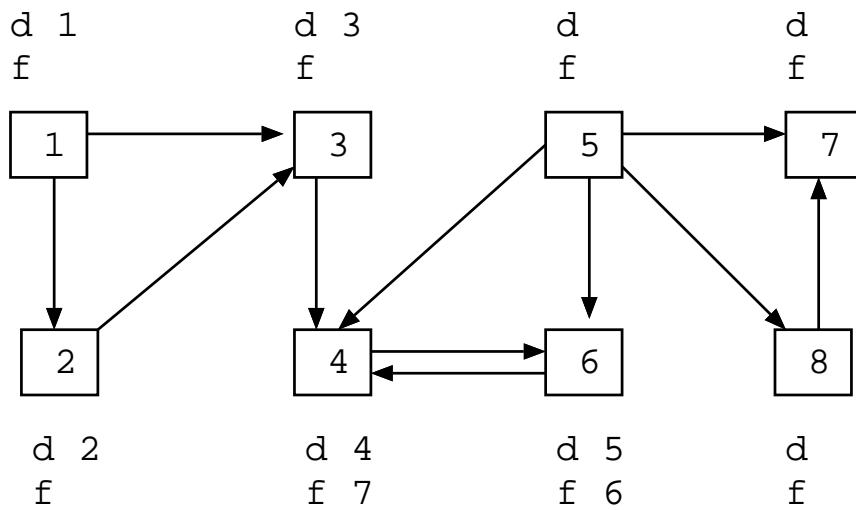
21-17: DFS Example



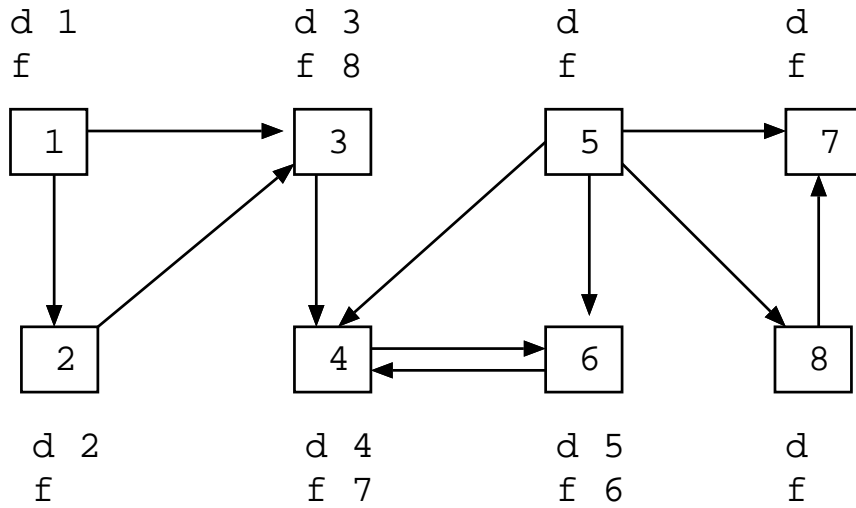
21-18: DFS Example



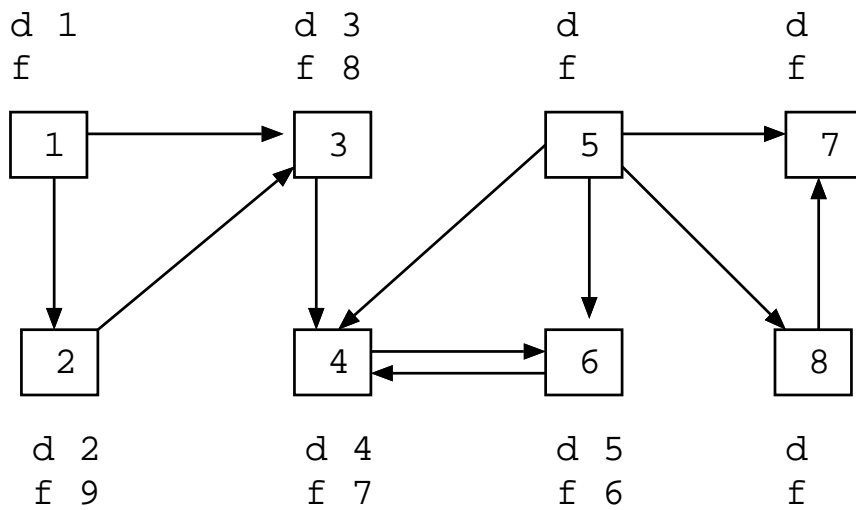
21-19: DFS Example



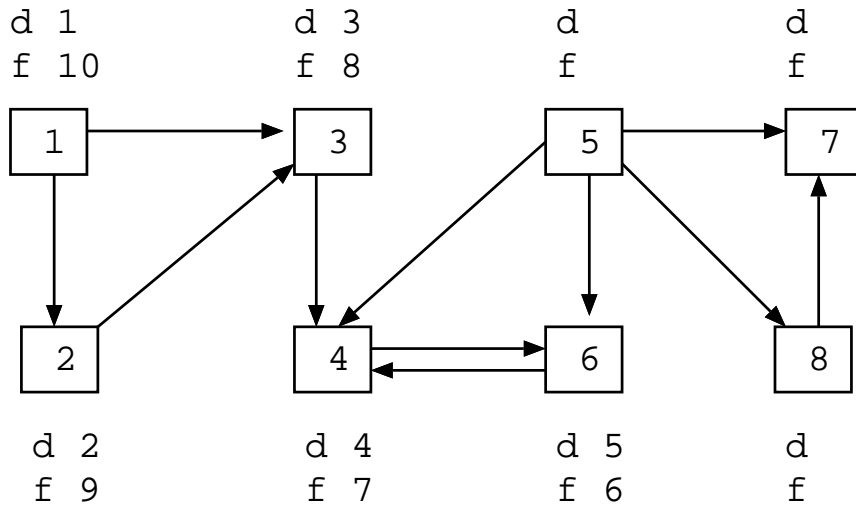
21-20: DFS Example



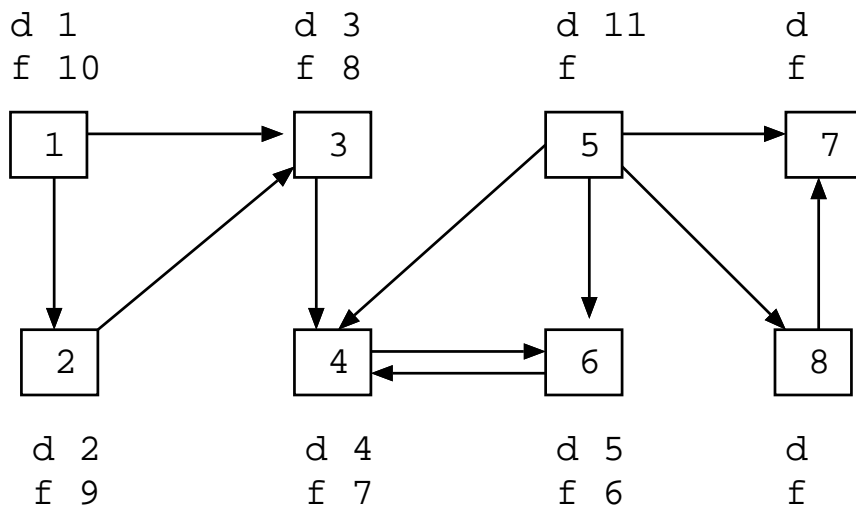
21-21: DFS Example



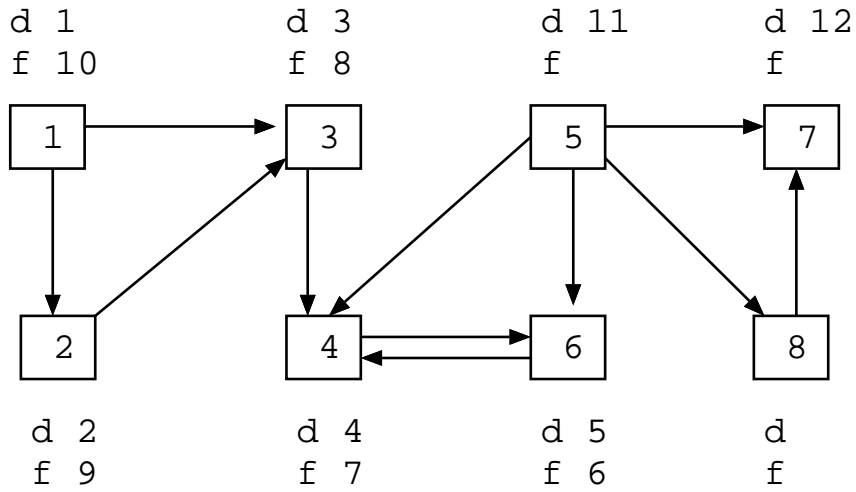
21-22: DFS Example



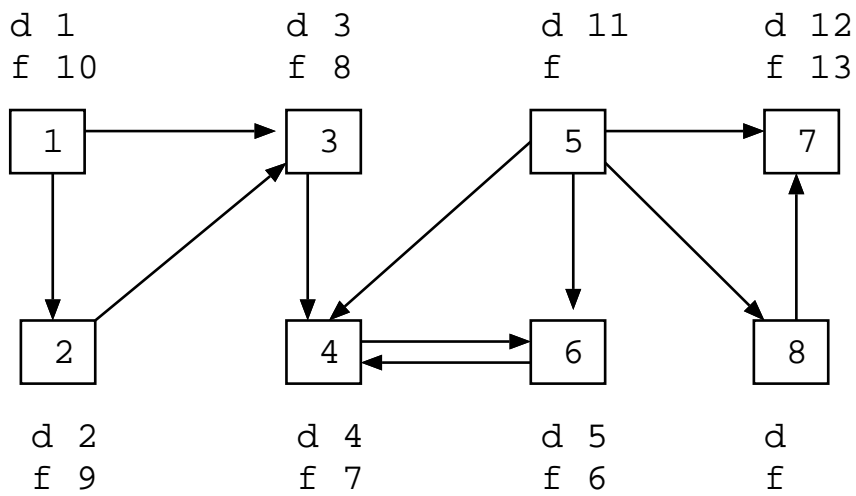
21-23: DFS Example



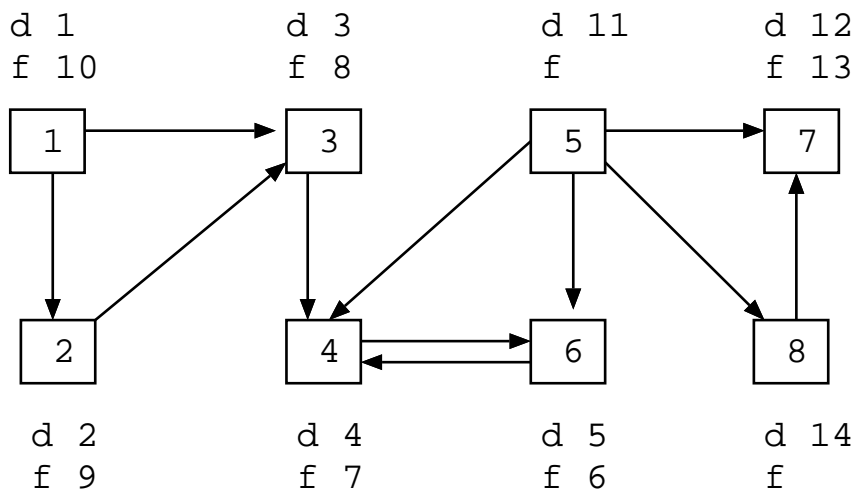
21-24: DFS Example



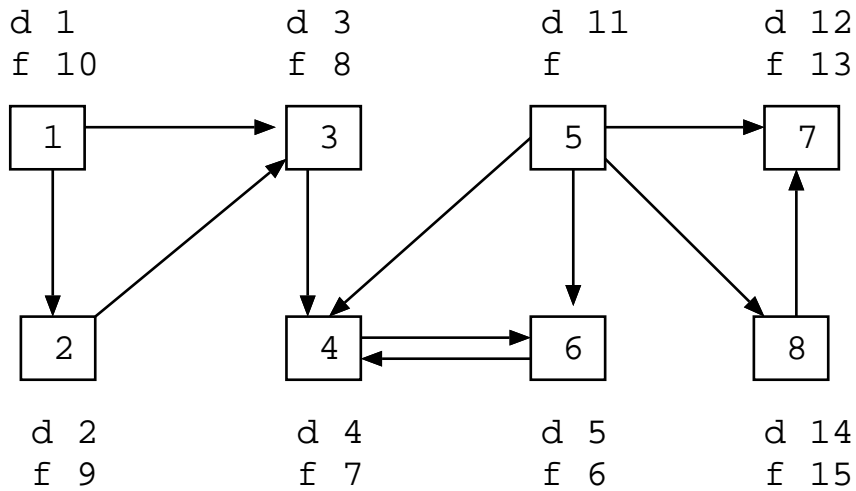
21-25: DFS Example



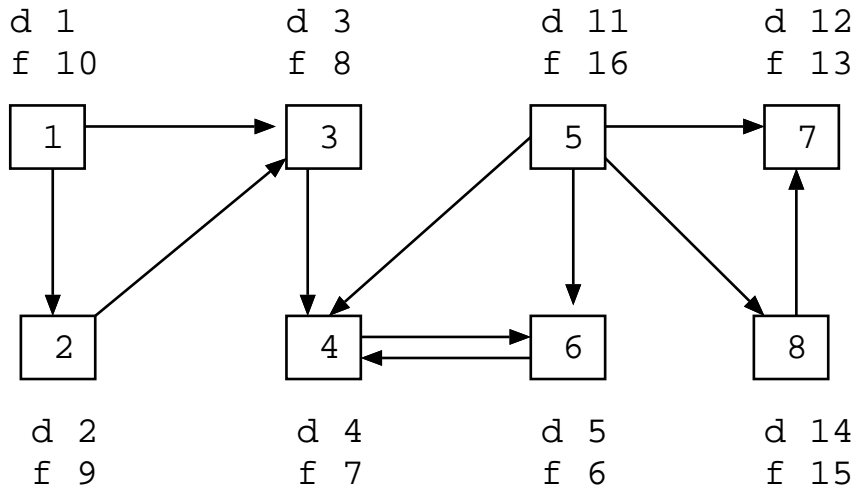
21-26: DFS Example



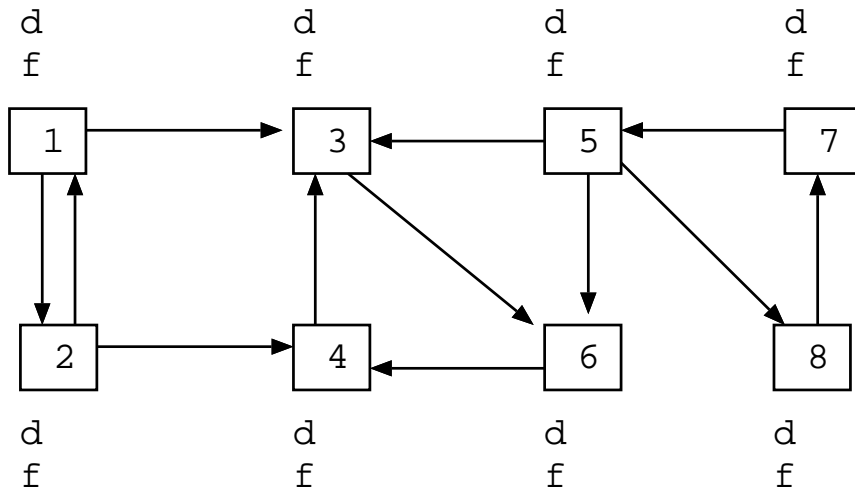
21-27: DFS Example



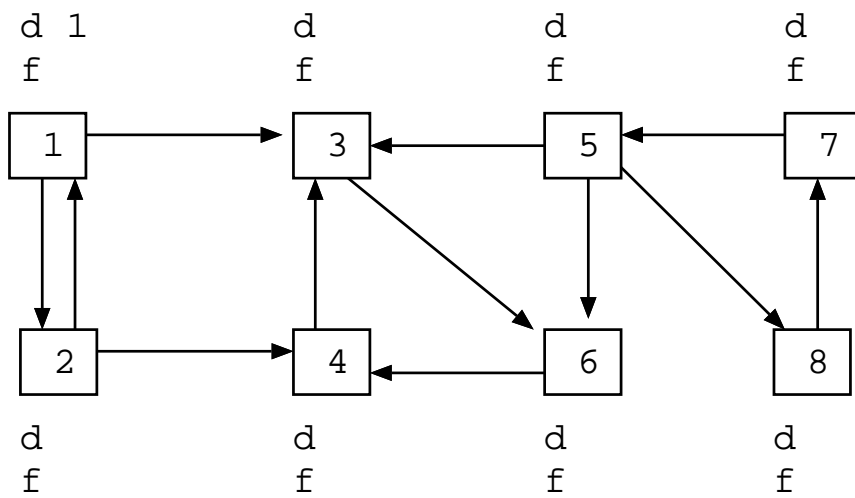
21-28: DFS Example



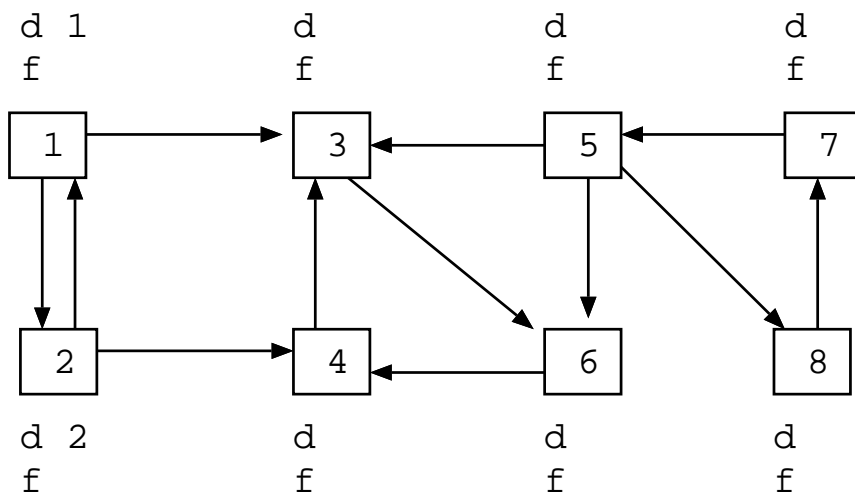
21-29: DFS Example



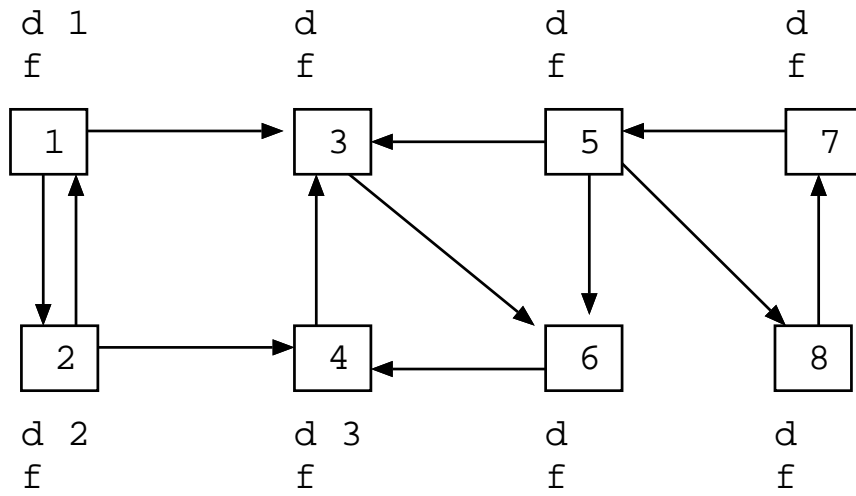
21-30: DFS Example



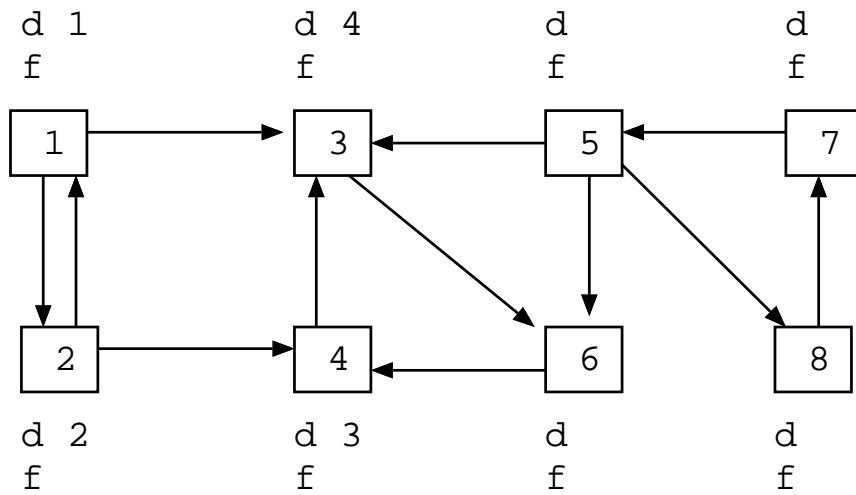
21-31: DFS Example



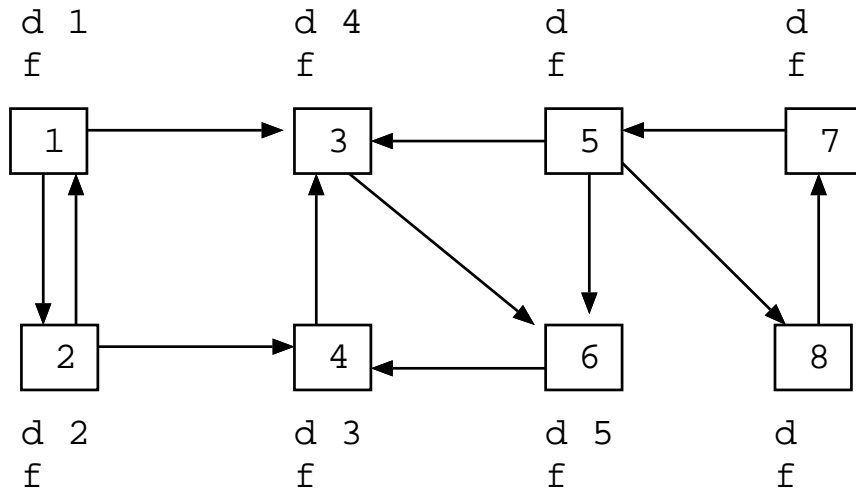
21-32: DFS Example



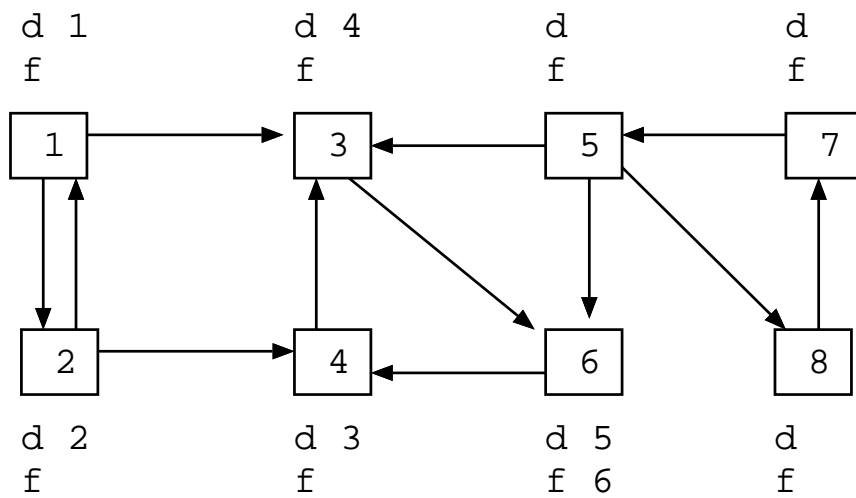
21-33: DFS Example



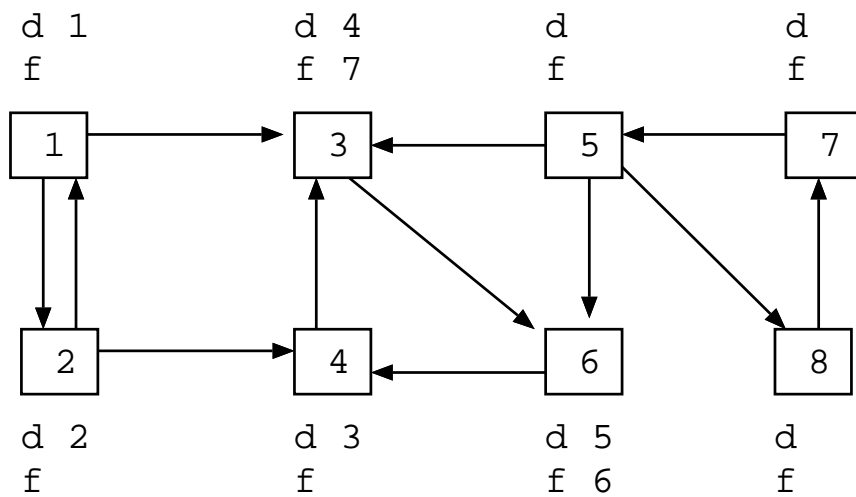
21-34: DFS Example



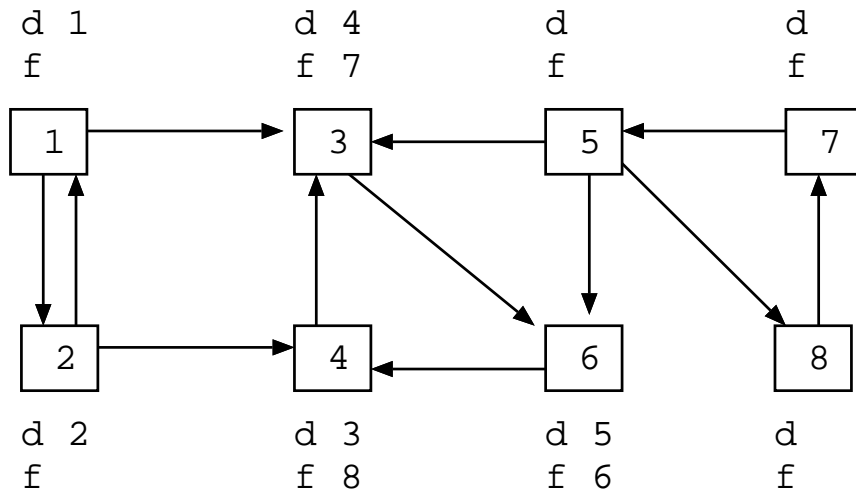
21-35: DFS Example



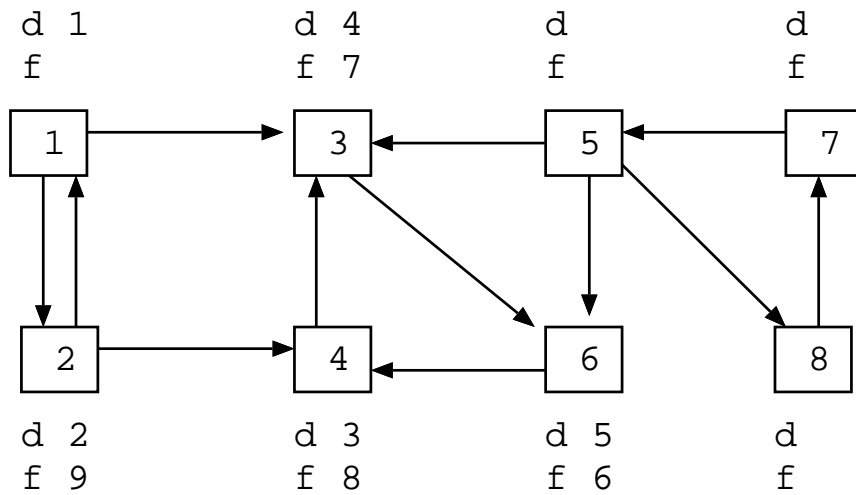
21-36: DFS Example



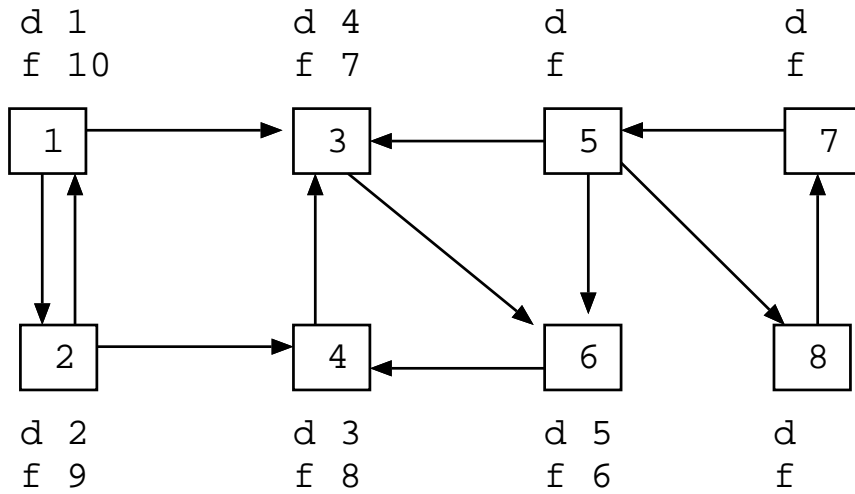
21-37: DFS Example



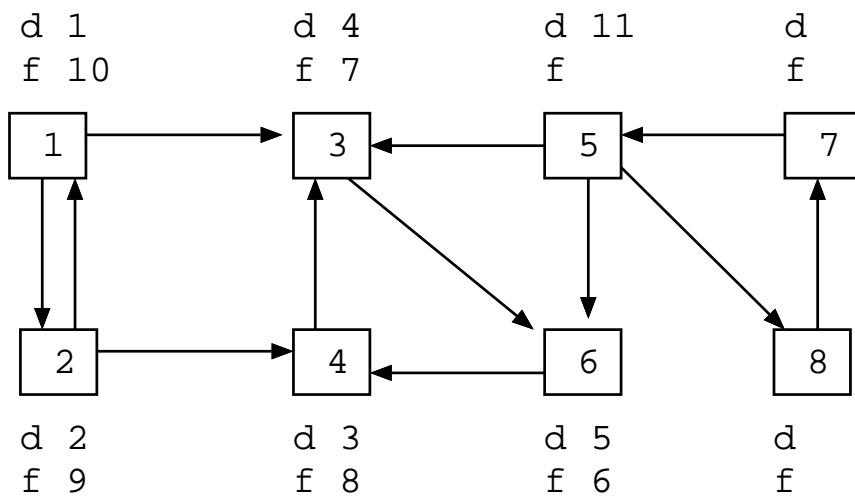
21-38: DFS Example



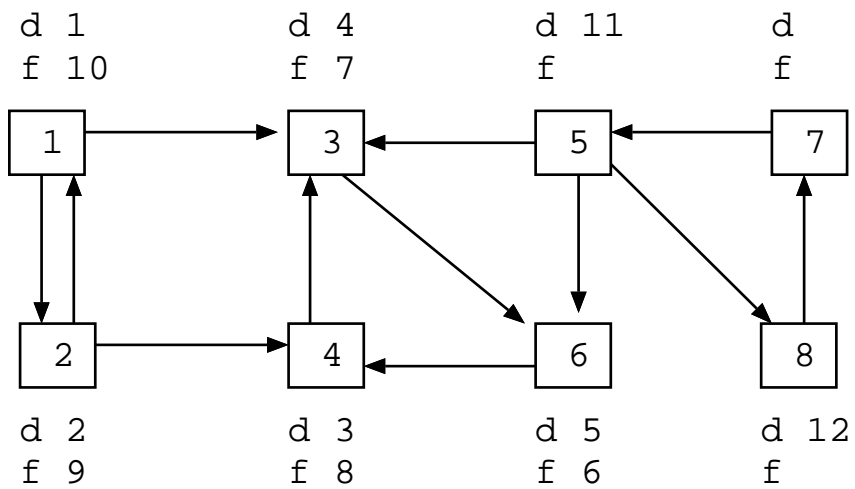
21-39: DFS Example



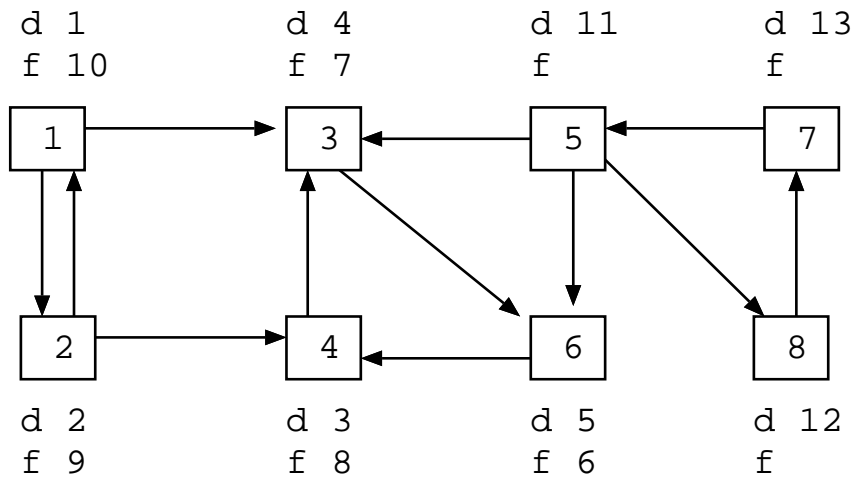
21-40: DFS Example



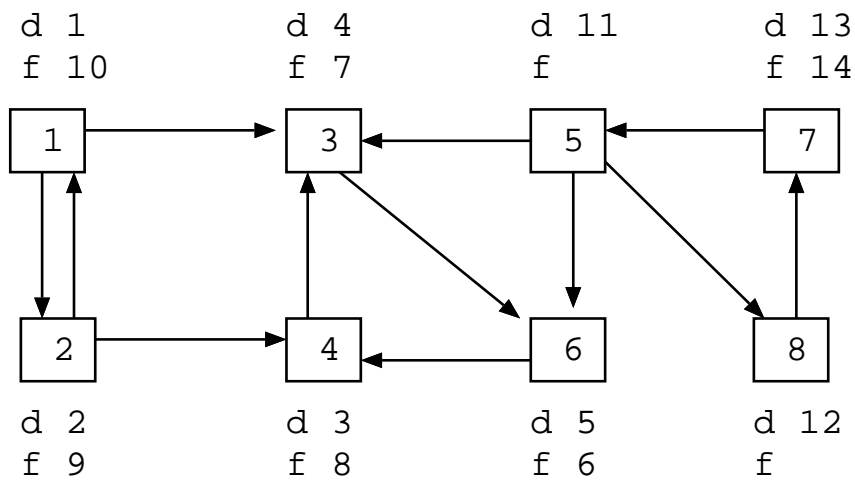
21-41: DFS Example



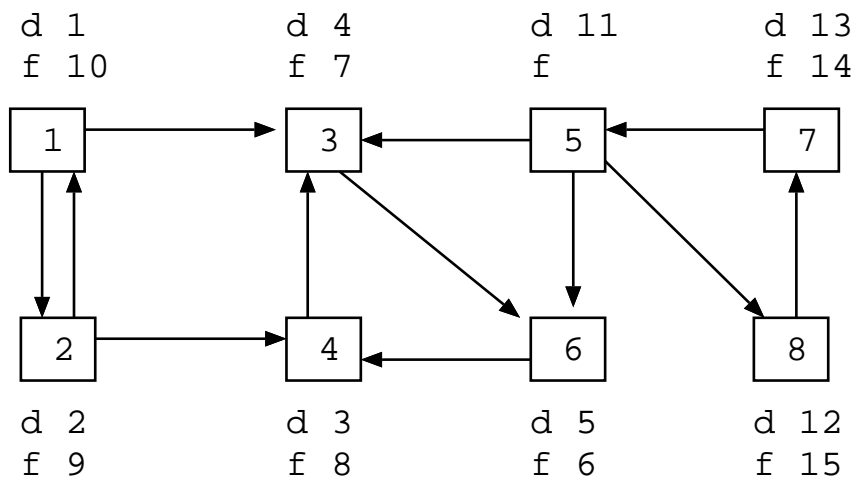
21-42: DFS Example



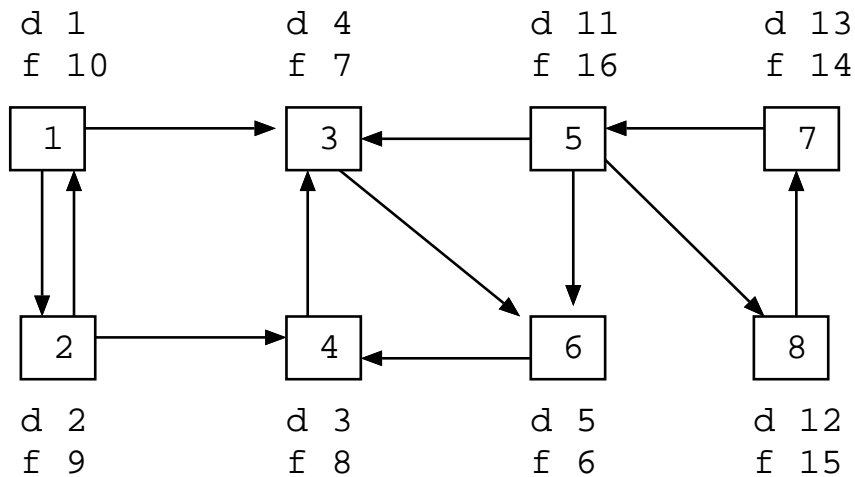
21-43: DFS Example



21-44: DFS Example



21-45: DFS Example



21-46: Using d[] & f[]

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

21-47: 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

21-48: 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

21-49: 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

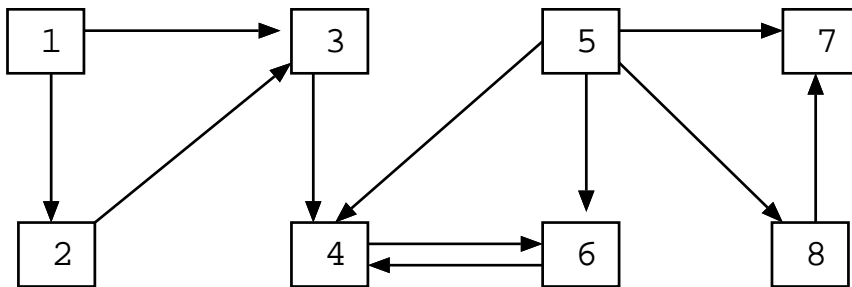
21-50: Calculating paths

- 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

21-51: Connected Components

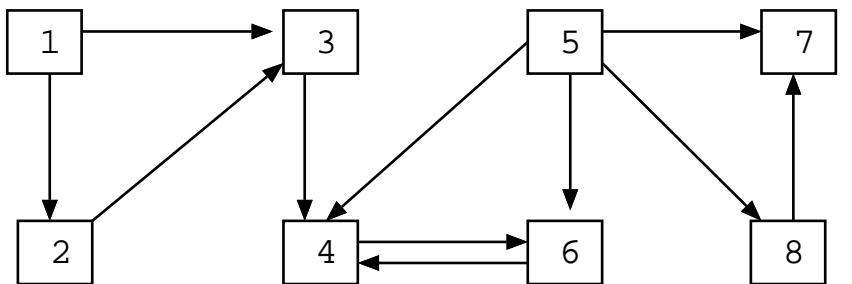
- Run DFS on G , calculating $f[]$ times
- Compute G^T
- Run DFS on $G^T -$ examining nodes in *inverse order of finishing times* from first DFS
- Any nodes that are in the same DFS search tree in G^T must be in the same connected component

21-52: Connected Components Eg.



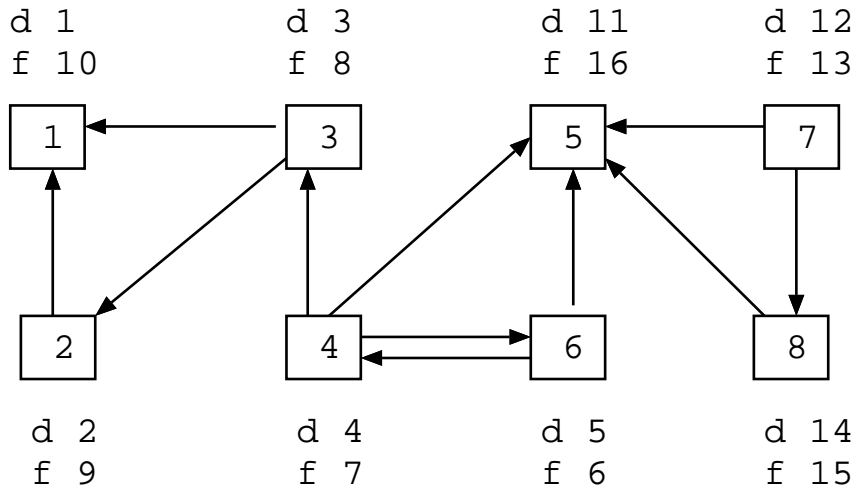
21-53: Connected Components Eg.

d 1	d 3	d 11	d 12
f 10	f 8	f 16	f 13

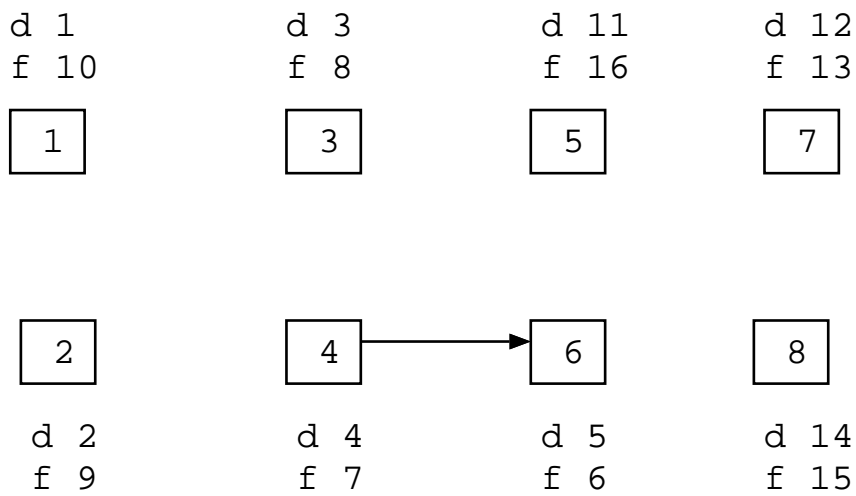


d 2	d 4	d 5	d 14
f 9	f 7	f 6	f 15

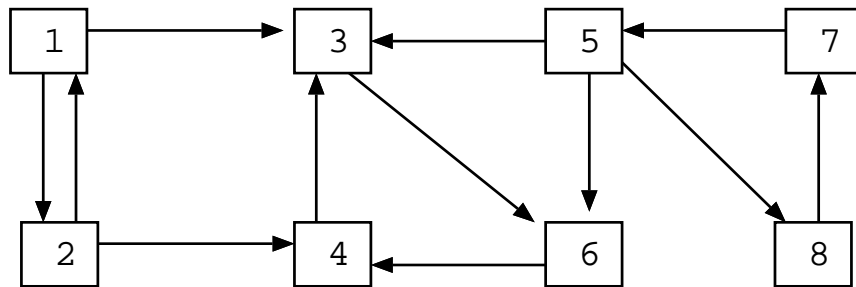
21-54: Connected Components Eg.



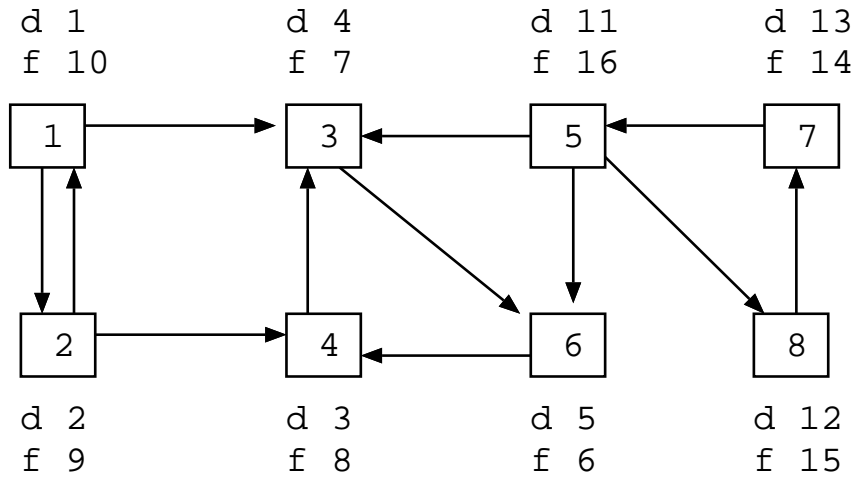
21-55: Connected Components Eg.



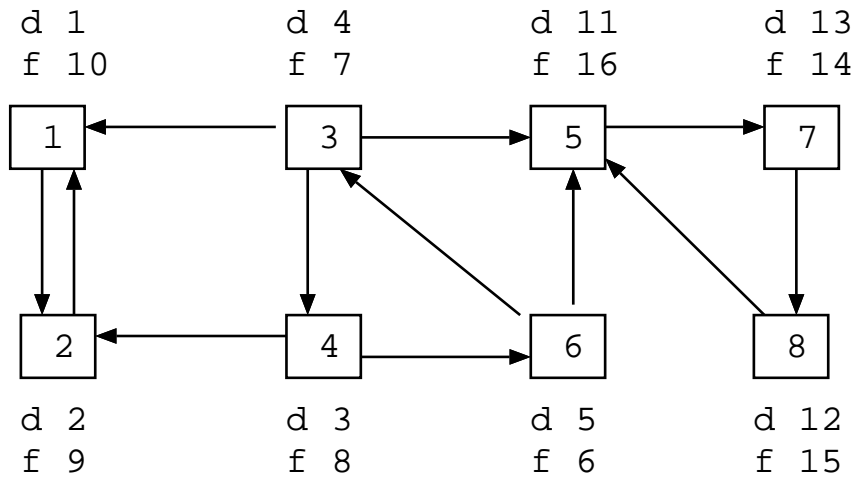
21-56: Connected Components Eg.



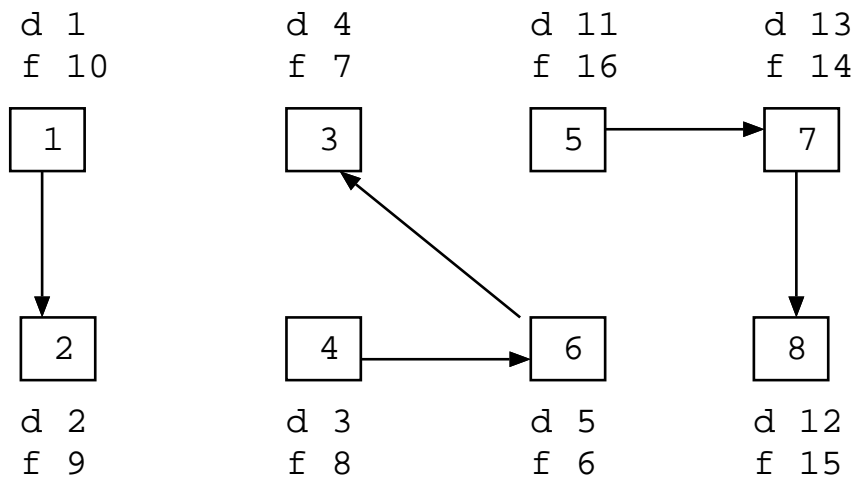
21-57: Connected Components Eg.



21-58: Connected Components Eg.



21-59: Connected Components Eg.



21-60: Topological Sort

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

21-61: **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 ?)

21-62: **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