19-0: **Strongly Connected Graph**

- Directed Path from every node to every other node

![Graph 1](image1)

- Strongly Connected

19-1: **Strongly Connected Graph**

- Directed Path from every node to every other node

![Graph 2](image2)

- Strongly Connected

19-2: **Connected Components**

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

![Graph 3](image3)

19-3: **Connected Components**
• Subgraph (subset of the vertices) that is strongly connected.

19-4: Connected Components

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

19-5: Connected Components

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

19-6: Connected Components

• Connected components of the graph are the \textit{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

19-7: 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

19-8: 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)

19-9: DFS Revisited

```java
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, d, f);
    }
    f[vertex] = time++;
}
```

19-10: DFS Revisited

- To visit every node in the graph:

```java
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);
}
```

19-11: DFS Example
19-12: DFS Example

19-13: DFS Example

19-14: DFS Example
19-15: DFS Example

19-16: DFS Example
19-17: DFS Example

19-18: DFS Example
19-19: DFS Example

19-20: DFS Example
19-21: DFS Example

19-22: DFS Example
19-23: DFS Example

19-24: DFS Example
19-25: DFS Example

19-26: DFS Example
19-27: DFS Example

19-28: DFS Example

19-29: DFS Example
19-30: DFS Example

d 1
d 2

d 3
d 4

d 5
d 6

d 7
d 8

19-31: DFS Example

d 1
d 2

d 3
d 4

d 5
d 6

d 7
d 8

19-32: DFS Example

19-33: DFS Example

19-34: DFS Example
19-35: DFS Example

19-36: DFS Example
19-37: DFS Example

19-38: DFS Example

19-39: DFS Example
19-40: DFS Example

19-41: DFS Example
19-42: DFS Example

19-43: DFS Example

19-44: DFS Example
19-45: **DFS Example**

\[
\begin{array}{cccc}
    d 1 & d 4 & d 11 & d 13 \\
    f 10 & f 7 & f 16 & f 14 \\
\end{array}
\]

19-46: **Using \(d[]\) & \(f[]\)**

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

19-47: **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 \(v_1\)

19-48: **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 \(v_1\)

19-49: **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

19-50: 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

19-51: 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

19-52: Connected Components Eg.

19-53: Connected Components Eg.

19-54: Connected Components Eg.
19-55: Connected Components Eg.

19-56: Connected Components Eg.

19-57: Connected Components Eg.
19-58: Connected Components Eg.

19-59: Connected Components Eg.

19-60: Topological Sort
• How could we use DFS to do a Topological Sort?
  • (Hint – Use discover and/or finish times)

19-61: **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 $y$?)

19-62: **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