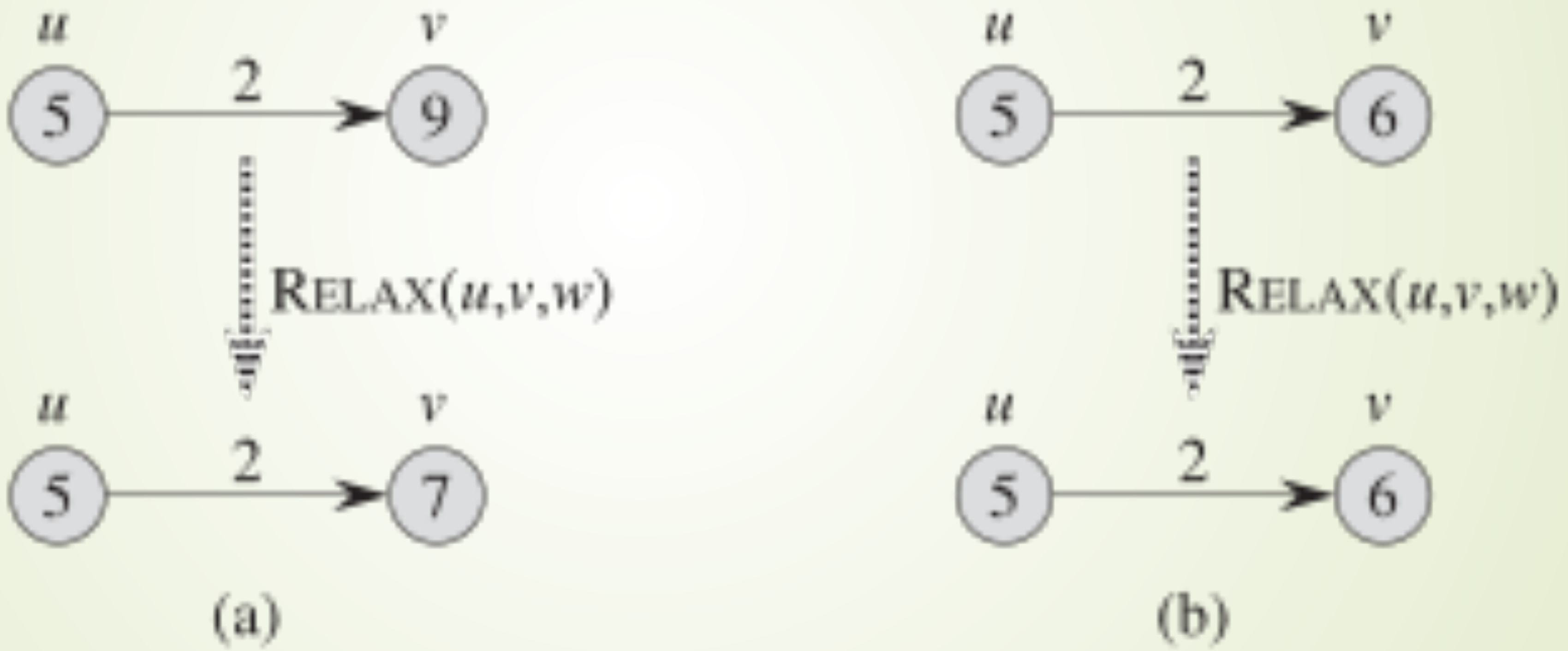
COT 6405: Analysis of Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/6405F19.html

Relax Step



All Pairs Shortest Path Algorithm

- Invoke Dijkstra's SSSP algorithm n times.
- Or use dynamic programming. How?

First Variant

- Let D[i,j,m] = length of the shortest path from I to j that uses at most m edges
- D[i,j,0] = ?; D[i,j,1] = ?
- Recurrence Relation

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j\ , \\ \infty & \text{if } i \neq j\ . \end{cases}$$

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} \right) \\ &= \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} . \end{aligned}$$

Second Variant

- C[i,j,k] = length of shortest path from i to j that only uses vertices from {1, 2, …, k}

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

Figure 14.38
Worst-case running times of various graph algorithms

Type of Graph Problem	Running Time	Comments
Unweighted	$O(\|E\|)$	Breadth-first search
Weighted, no negative edges	$O(\|E\| \log \|V\|)$	Dijkstra's algorithm
Weighted, negative edges	$O(\|E\| \cdot \|V\|)$	Bellman–Ford algorithm
Weighted, acyclic	$O(\|E\|)$	Uses topological sort

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

All Pairs Shortest Path

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

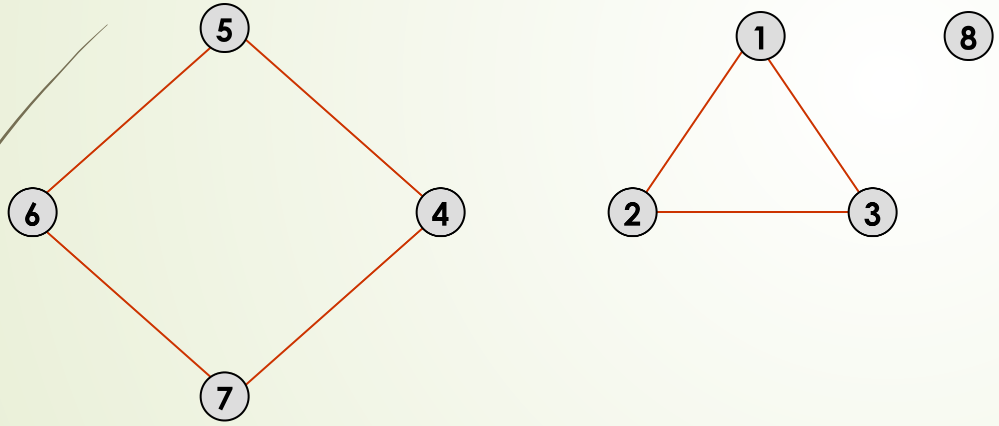
Main loops of Floyd-Warshall's algorithm

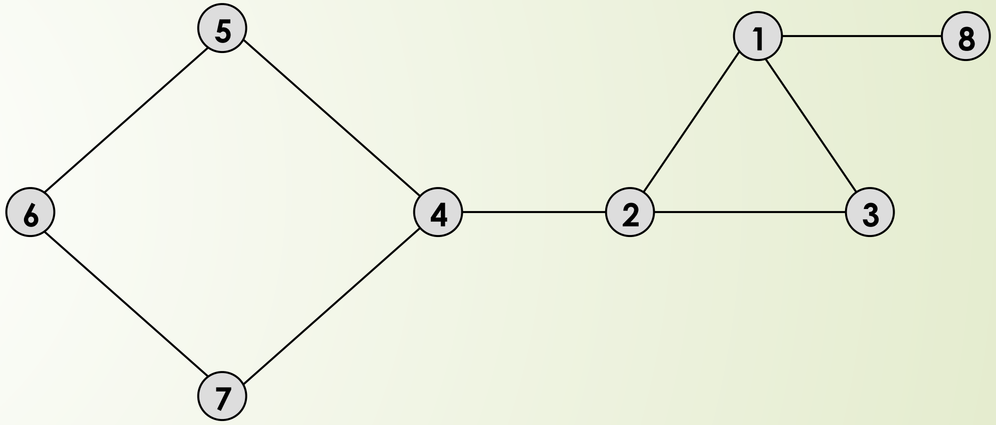
```
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do if c_ij > c_ik + c_kj
                then c_ij ← c_ik + c_kj
```

Time Complexity

- Time Complexity = $O(n^3)$
- Improvements are possible with faster matrix multiplication algorithm.

Connectivity & Biconnectivity





Connectivity & Biconnectivity: Undirected Case

- Graph is **connected** if there exists a path between every pair of vertices.
- A tree is **minimally connected**
- Removing a edge/vertex from a **minimally connected** graph makes it disconnected.

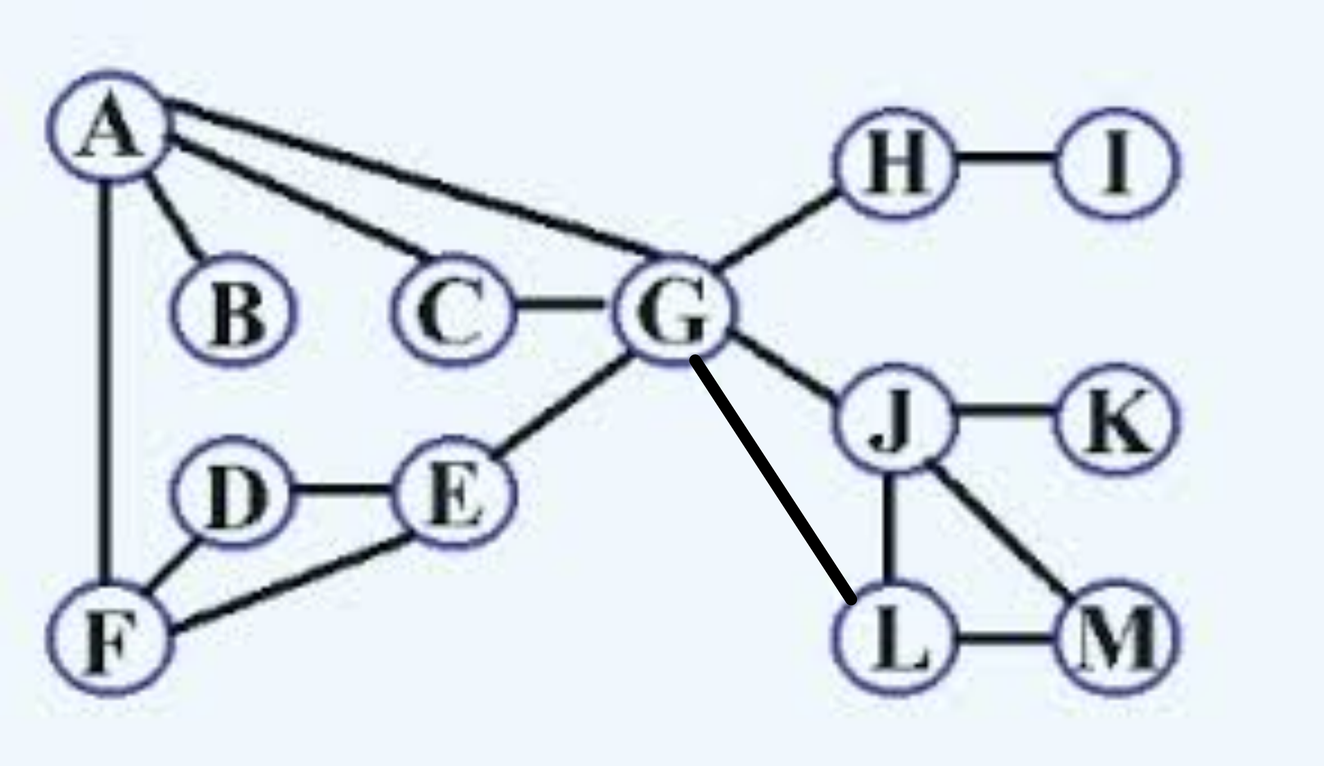
- Graph is **biconnected** if there exists 2 or more **disjoint** paths between every pair of vertices.
- A cycle is **minimally biconnected**
- You need to remove at least 2 vertices/edges to disconnect a **minimally biconnected** graph.
- Every node lies on a cycle

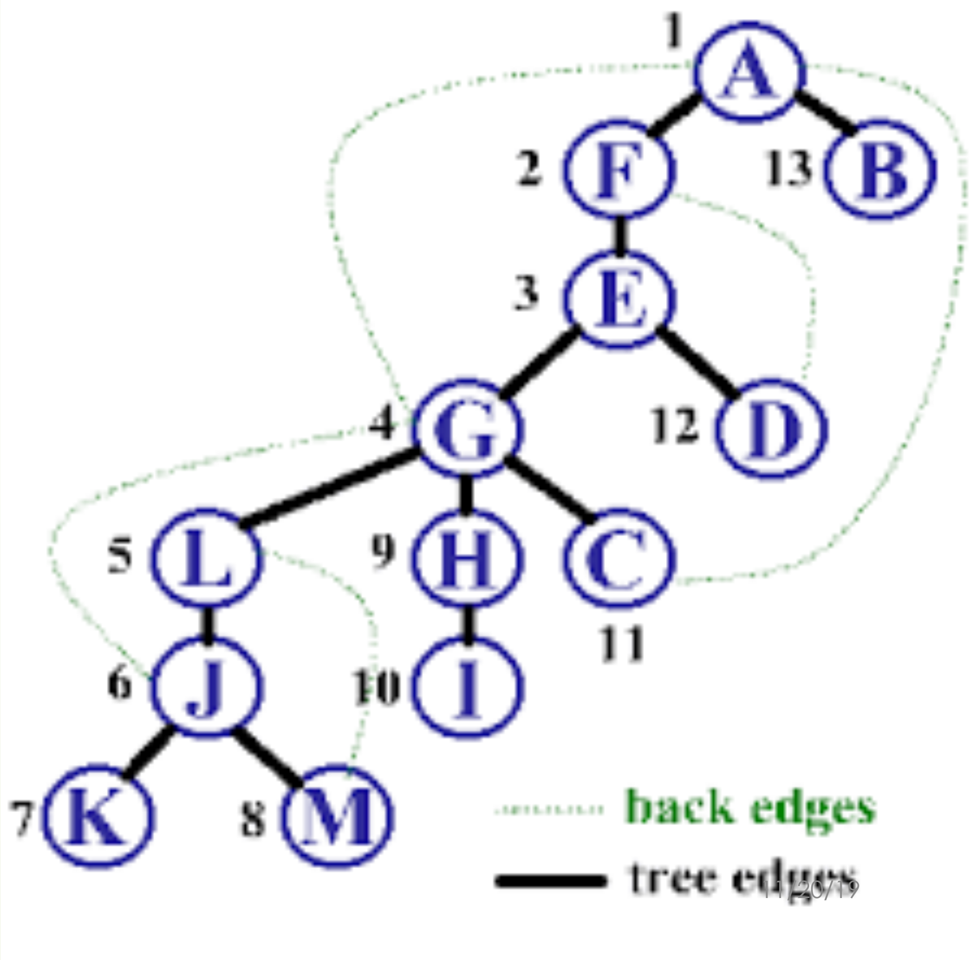
Connected & Biconnected Components

- Subgraph G'(V',E') is a **connected component** of G(V,E) if V' is a maximal subset of V that induces a connected subgraph.
- If a graph is not connected, it can be decomposed into connected components.

- Subgraph G'(V',E') is a **biconnected component** of G(V,E) if V' is a maximal subset of V that induces a biconnected subgraph.
- If a graph is not biconnected, it can be decomposed into biconnected components.

What does DFS do for us?





Testing for Biconnectivity

- An **articulation point** is a vertex whose removal disconnects graph.
- A **bridge** is an edge whose removal disconnects graph.
- **Claim:** If a graph is not biconnected, it must have an articulation point. **Proof**? "If and only if"?
- How do we look for articulation points (and bridges)?
  - Use DFS

Biconnectivity Principles

- If root of DFS tree has at least 2 children, it's an articulation point
  - Easy to check!
- Non-root vertex u is an articulation point of G if and only if u has a child v such that there is no back edge from v or any descendant of v to a proper ancestor of u
- Compute Low[x] = lowest numbered vertex reachable from some descendant of x (default is d[x])
- Vertex u is an articulation point if Low[s] >= d[u] for child s of u

BCC

```
DFS-VISIT(u)
VisitVertex(u)
Color[u] ← GRAY
Time ← Time + 1
d[u] ← Time
for each v ∈ Adj[u] do
    VisitEdge(u,v)
    if (v ≠ π[u]) then
      if (color[v] = WHITE) then
            π[v] ← u
            DFS-VISIT(v)
color[u] ← BLACK
F[u] ← Time ← Time + 1
```

```
BCC(G, u) // Compute the biconnected components of G
    // starting from vertex u
Color[u] ← GRAY
Low[u] ← d[u] ← Time ← Time + 1
Put u on stack S
for each v ∈ Adj[u] do
    if (v ≠ π[u]) and (color[v] ≠ BLACK) then
        if (TopOfStack(S) ≠ u) then put u on stack S
        Put edge (u,v) on stack S
        if (color[v] = WHITE) then
          π[v] ← u
            BCC(G, v)
            if (Low[v] >= d[u]) then // u is an articul. pt.
                // Output next biconnected component
                Pop S until u is reached
                Push u back on S
            Low[u] = min { Low[u], Low[v] }
        else Low[u] = min { Low[u], d[v] } // back edge
```

Correctness and Complexity

- Theorem: A graph is biconnected if and only if it has no articulation points
- BCC finds all articulation points
  - If Low[child(u)] >= u, then u is an articulation point
- Correctness follows from theoretical principles
- Time and Space complexity = O(n+m) **Why**?

How to detect bridges

- An edge e of G is a bridge if and only if it does not lie on any simple cycle of G
  - Use DFS, where every edge is a tree edge or back edge
  - If edge e is a back edge?
    - It cannot be a bridge! **Why**?
  - If edge e is a tree edge?
    - Let e = (u,v) such that u is the parent of v
    - Edge e is a bridge if Low[v] = d[v]

Correctness and Complexity

- Correctness follows from the theoretical principles
- Time and Space complexity to detect all bridges in the graph
  - O(n+m) **Why**?